(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
12 July 2001 (12.07.2001)

PCT

(10) International Publication Number
WO 01/50679 A2

(51) International Patent Classification⁷: H04L 12/00

(21) International Application Number: PCT/US00/33405

(22) International Filing Date: 7 December 2000 (07.12.2000)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
09/474,650 29 December 1999 (29.12.1999) US

(63) Related by continuation (CON) or continuation-in-part (CIP) to earlier application:
US 09/474,650 (CON)
Filed on 29 December 1999 (29.12.1999)

(71) Applicant *(for all designated States except US)*: INTEL CORPORATION [US/US]; 2200 Mission College Boulevard, Santa Clara, CA 95052 (US).

(72) Inventors; and
(75) Inventors/Applicants *(for US only)*: WOLRICH, Gilbert [US/US]; 4 Cider Mill Road, Framingham, MA 01701 (US). BERNSTEIN, Debra [US/US]; 443 Peakham Road, Sudbury, MA 01776 (US). ADILETTA, Matthew, J. [US/US]; 20 Monticello Drive, Worcester, MA 01603 (US). HOOPER, Donald, F. [US/US]; 19 Main Circle, Shrewsbury, MA 01545 (US).

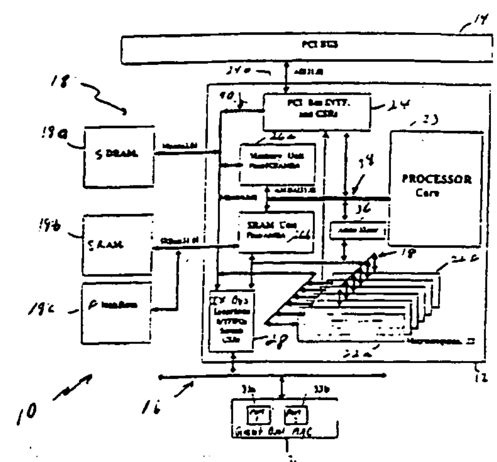
(74) Agent: HARRIS, Scott, C.; Fish & Richardson P.C., 4350 La Jolla Village Drive, Suite 500, San Diego, CA 92122 (US).

(81) Designated States *(national)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CR, CU, CZ, DE, DK, DM, DZ, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, TZ, UA, UG, US, UZ, VN, YU, ZA, ZW.

(84) Designated States *(regional)*: ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE, TR), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GW, ML, MR, NE, SN, TD, TG).

[Continued on next page]

(54) Title: METHOD AND APPARATUS FOR GIGABIT PACKET ASSIGNMENT FOR MULTITHREADED PACKET PROCESSING



(57) Abstract: A network processor that has multiple processing elements, each supporting multiple simultaneous program threads with access to shared resources in an interface. Packet data is received from high-speed ports in segments and each segment is assigned to one of the program threads. Each packet may be assigned to a single program thread, two program threads - one for header segment processing and the other for handling payload segment(s) - or a different program thread for segment of data in a packet. Dedicated inputs for ready status and sequence numbers provide assistance needed for receiving the packet data over a high speed port. The dedicated inputs are used to monitor ready flags from the high speed ports on a cycle-by-cycle basis. The sequence numbers are used by the assigned threads to maintain ordering of segments within a packet, as well as to order the writes of the complete packets to transmit queues.

WO 01/50679 A2

Published:
— *Without international search report and to be republished upon receipt of that report.*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

METHOD AND APPARATUS FOR GIGABIT PACKET ASSIGNMENT FOR MULTITHREADED PACKET PROCESSING

Background of the Invention

The invention relates generally to network data processing.

Networking products such as routers require high speed components for packet data movement, i.e., collecting packet data from incoming network device ports and queuing the packet data for transfer to appropriate forwarding device ports. They also require high-speed special controllers for processing the packet data, that is, parsing the data and making forwarding decisions. Because the implementation of these high-speed functions usually involves the development of ASIC or custom devices, such networking products are of limited flexibility and thus tend to be quite rigid in their assignment of ports to the high-speed controllers. Typically, each controller is assigned to service network packets from for one or more given ports on a permanent basis.

Summary of the Invention

In one aspect of the invention, forwarding data includes associating control information with data received from a first port and using the associated control information to enqueue the data for transmission to a second port in the same order in which the data was received from the first port.

Brief Description of the Drawings

Other features and advantages of the invention will be apparent from the following description taken together with the drawings in which:

FIG. 1 is a block diagram of a communication system employing a hardware-based multi-threaded processor;

FIG. 2 is a block diagram of a microengine employed in the hardware-based multi-threaded processor of FIG. 1;

FIG. 3 is an illustration of an exemplary thread task assignment;

FIG. 4 is a block diagram of an I/O bus interface shown in FIG. 1;

FIG. 5 is a detailed diagram of a bus interface unit

employed by the I/O bus interface of FIG. 4;

FIGS. 6A-6F are illustrations of various bus configuration control and status registers (CSRs);

FIG. 7 is a detailed diagram illustrating the interconnection between two Gigabit Ethernet ("fast") ports and the bus interface unit;

FIGS. 8A-8C are illustrations of the formats of the RCV_RDY_CTL, RCV_RDY_HI and RCV_RDY_LO CSR registers, respectively;

FIG. 9 is a depiction of the receive threads and their interaction with the I/O bus interface during a receive process;

FIGS. 10A and 10B are illustrations of the format of the RCV_REQ FIFO and the RCV_CTL FIFO, respectively;

FIGS. 11A-11B are illustrations of the formats of the SOP_SEQx registers and ENQUEUE_SEQx registers, respectively;

FIG. 12 is a flow diagram of the receive process for fast ports;

FIGS. 13A and 13B are flow diagrams which illustrate portions of the receive process for fast ports using a single thread mode;

FIGS. 14A and 14B are flow diagrams which illustrate

portions of the receive process for fast ports using a dual thread (or header/body) mode;

FIGS. 15A and 15B are flow diagrams which illustrate portions of the receive process for fast ports using an explicit (distributed thread) mode; and

FIG. 16 is a flow diagram of a packet enqueuing process for fast ports.

Detailed Description

Referring to FIG. 1, a communication system 10 includes a parallel, hardware-based multi-threaded processor 12. The hardware based multi-threaded processor 12 is coupled to a first peripheral bus (shown as a PCI bus) 14, a second peripheral bus referred to as an I/O bus 16 and a memory system 18. The system 10 is especially useful for tasks that can be broken into parallel subtasks or functions. The hardware-based multi-threaded processor 12 includes multiple microengines 22, each with multiple hardware controlled program threads that can be simultaneously active and independently work on a task. In the embodiment shown, there are six microengines 22a-22f and each of the six microengines is capable of processing

four program threads, as will be described more fully below.

The hardware-based multi-threaded processor 12 also includes a processor 23 that assists in loading microcode control for other resources of the hardware-based multi-threaded processor 12 and performs other general purpose computer type functions such as handling protocols, exceptions, extra support for packet processing where the microengines pass the packets off for more detailed processing. In one embodiment, the processor 23 is a StrongARM (ARM is a trademark of ARM Limited, United Kingdom) core based architecture. The processor (or core) 23 has an operating system through which the processor 23 can call functions to operate on the microengines 22a-22f. The processor 23 can use any supported operating system, preferably real-time operating system. For the core processor implemented as a StrongARM architecture, operating systems such as MicrosoftNT real-time, VXWorks and :CUS, a freeware operating system available over the Internet, can be used.

The six microengines 22a-22f each operate with shared resources including the memory system 18, a PCI bus interface 24 and an I/O bus interface 28. The PCI bus

interface provides an interface to the PCI bus 14. The I/O bus interface 28 is responsible for controlling and interfacing the processor 12 to the I/O bus 16. The memory system 18 includes a Synchronous Dynamic Random Access Memory (SDRAM) 18a, which is accessed via an SDRAM controller 26a, a Static Random Access Memory (SRAM) 18b, which is accessed using an SRAM controller 26b, and a nonvolatile memory (shown as a FlashROM) 18c that is used for boot operations. The SDRAM 16a and SDRAM controller 26a are typically used for processing large volumes of data, e.g., processing of payloads from network packets. The SRAM 18b and SRAM controller 26b are used in a networking implementation for low latency, fast access tasks, e.g., accessing look-up tables, memory for the processor 23, and so forth. The microengines 22a-22f can execute memory reference instructions to either the SDRAM controller 26a or the SRAM controller 18b.

The hardware-based multi-threaded processor 12 interfaces to network devices such as a media access controller device, including a high-speed (or fast) device 31, such as Gigabit Ethernet MAC, ATM device or the like, over the I/O bus 16. In the embodiment shown, the high-speed device is a Dual Gigabit MAC device having two fast

ports 33a, 33b. Each of the network devices attached to the I/O bus 16 can include a plurality of ports to be serviced by the processor 12. Other devices, such as a host computer (not shown), that may be coupled to the PCI bus 14 are also serviced by the processor 12. In general, as a network processor, the processor 12 can interface to any type of communication device or interface that receives/sends large amounts of data. The processor 12 functioning as a network processor could receive units of packet data from the device 31 and process those units of packet data in a parallel manner, as will be described. The unit of packet data could include an entire network packet (e.g., Ethernet packet) or a portion of such a packet.

Each of the functional units of the processor 12 are coupled to one or more internal buses. The internal buses include an internal core bus 34 (labeled "AMBA") for coupling the processor 23 to the memory controllers 26a, 26b and to an AMBA translator 36. The processor 12 also includes a private bus 38 that couples the microengines 22a-22f to the SRAM controller 26b, AMBA translator 36 and the Fbus interface 28. A memory bus 40 couples the memory controllers 26a, 26b to the bus interfaces 24, 28 and the

memory system 18.

Referring to FIG. 3, an exemplary one of the microengines 22a-22f is shown. The microengine 22a includes a control store 70 for storing a microprogram. The microprogram is loadable by the central processor 20. The microengine 70 also includes control logic 72. The control logic 72 includes an instruction decoder 73 and program counter units 72a-72d. The four program counters are maintained in hardware. The microengine 22a also includes context event switching logic 74. The context event switching logic 74 receives messages (e.g., SEQ_#_EVENT_RESPONSE; FBI_EVENT_RESPONSE; SRAM_EVENT_RESPONSE; SDRAM_EVENT_RESPONSE; and AMBA_EVENT_RESPONSE) from each one of the share resources, e.g., SRAM 26b, SDRAM 26a, or processor core 20, control and status registers, and so forth. These messages provides information on whether a requested function has completed. Based on whether or not the function requested by a thread has completed and signaled completion, the thread needs to wait for that complete signal, and if the thread is enable to operate, then the thread is place on an available thread list (not shown). As earlier mentioned, the microengine 22a can have a maximum of 4 threads of

execution available.

In addition to event signals that are local to an executing thread, the microengine employs signaling states that are global. With signaling states, an executing thread can broadcast a signal state to all microengines 22. Any and all threads in the microengines can branch on these signaling states. These signaling states can be used to determine availability of a resource or whether a resource is due for servicing.

The context event logic 74 has arbitration for the four threads. In one embodiment, the arbitration is a round robin mechanism. However, other arbitration techniques, such as priority queuing or weighted fair queuing, could be used. The microengine 22a also includes and execution box (EBOX) data path 76 that includes an arithmetic logic unit (ALU) 76a and a general purpose register (GPR) set 76b. The ALU 76a performs arithmetic and logical functions as well as shift functions.

The microengine 22a further includes a write transfer registers file 78 and a read transfer registers file 80. The write transfer registers file 78 stores data to be written to a resource. The read transfer registers file 80 is for storing return data from a resource.

Subsequent to or concurrent with the data arrival, an event signal from the respective shared resource, e.g., memory controllers 26a, 26b, or core 23, will be provided to the context event arbiter 74, which in turn alerts the thread that the data is available or has been sent. Both transfer register files 78, 80 are connected to the EBOX 76 through a data path. In the described implementation, each of the register files includes 64 registers.

The functionality of the microengine threads is determined by microcode loaded (via the core processor) for a particular user's application into each microengine's control store 70. Referring to FIG. 3, an exemplary thread task assignment 90 is shown. Typically, one of the microengine threads is assigned to serve as a receive scheduler 92 and another as a transmit scheduler 94. A plurality of threads are configured as receive processing threads 96 and transmit processing (or "fill") threads 98. Other thread task assignments include a transmit arbiter 100 and one or more core communication threads 102. Once launched, a thread performs its function independently.

The receive scheduler thread 92 assigns packets to receive processing threads 96. In a packet forwarding application for a bridge/router, for example, the receive

processing thread parses packet headers and performs lookups based in the packet header information. Once the receive processing thread or threads 96 has processed the packet, it either sends the packet as an exception to be further processed by the core 23 (e.g., the forwarding information cannot be located in lookup and the core processor must learn it), or stores the packet in the SDRAM and queues the packet in a transmit queue by placing a packet link descriptor for it in a transmit queue associated with the transmit (forwarding port) indicated by the header/lookup. The transmit queue is stored in the SRAM. The transmit arbiter thread 100 prioritizes the transmit queues and the transmit scheduler thread 94 assigns packets to transmit processing threads that send the packet out onto the forwarding port indicated by the header/lookup information during the receive processing.

The receive processing threads 96 may be dedicated to servicing particular ports or may be assigned to ports dynamically by the receive scheduler thread 92. For certain system configurations, a dedicated assignment may be desirable. For example, if the number of ports is equal to the number of receive processing threads 96, then it may be quite practical as well as efficient to assign the

receive processing threads to ports in a one-to-one, dedicated assignment. In other system configurations, a dynamic assignment may provide a more efficient use of system resources.

The receive scheduler thread 92 maintains scheduling information 104 in the GPRs 76b of the microengine within which it executes. The scheduling information 104 includes thread capabilities information 106, port-to-thread assignments (list) 108 and "thread busy" tracking information 110. At minimum, the thread capabilities information informs the receive scheduler thread as to the type of tasks for which the other threads are configured, e.g., which threads serve as receive processing threads. Additionally, it may inform the receive scheduler of other capabilities that may be appropriate to the servicing of a particular port. For instance, a receive processing thread may be configured to support a certain protocol, or a particular port or ports. A current list of the ports to which active receive processing threads have been assigned by the receive scheduler thread is maintained in the thread-to-port assignments list 108. The thread busy mask register 110 indicates which threads are actively servicing a port. The receive scheduler uses all of this scheduling

information in selecting threads to be assigned to ports that require service for available packet data, as will be described in further detail below.

Referring to FIG. 4, the I/O bus interface 28 includes shared resources 120, which are coupled to a push/pull engine interface 122 and a bus interface unit 124. The bus interface unit 124 includes a ready bus controller 126 connected to a ready bus 128 and an Fbus controller 130 for connecting to a portion of the I/O bus referred to as an Fbus 132. Collectively, the ready bus 128 and the Fbus 132 make up the signals of the I/O bus 16 (FIG. 1). The resources 120 include two FIFOs, a transmit FIFO 134 and a receive FIFO 136, as well as CSRs 138, a scratchpad memory 140 and a hash unit 142. The Fbus 132 transfers data between the ports of the device 31 and the I/O bus interface 28. The ready bus 128 is an 8-bit bus that performs several functions. It is used to read control information about data availability from the device 31, e.g., in the form of ready status flags. It also provides flow control information to the device 31 and may be used to communicate with another network processor 12 that is connected to the Fbus 132. Both buses 128, 132 are accessed by the microengines 22 through the CSRs 138. The

CSRs 138 are used for bus configuration, for accessing the bus interface unit 124, and for inter-thread signaling. They also include a several counters and thread status registers, as will be described. The CSRs 138 are accessed by the microengines 22 and the core 23. The receive FIFO (RFIFO) 136 includes data buffers for holding data received from the Fbus 132 and is read by the microengines 22. The transmit FIFO (TFIFO) 134 includes data buffers that hold data to be transmitted to the Fbus 132 and is written by the microengines 22. The scatchpad memory 140 is accessed by the core 23 and microengines 22, and supports a variety of operations, including read and write operations, as well as bit test, bit test/clear and increment operations. The hash unit 142 generates hash indexes for 48-bit or 64-bit data and is accessed by the microengines 22 during lookup operations.

The processors 23 and 22 issue commands to the push/pull engine interface 122 when accessing one of the resources 120. The push/pull engine interface 122 places the commands into queues (not shown), arbitrates which commands to service, and moves data between the resources 120, the core 23 and the microengines 22. In addition to servicing requests from the core 23 and microengines 22,

the push/pull engines 122 also service requests from the ready bus 128 to transfer control information to a register in the microengine read transfer registers 80.

When a thread issues a request to a resource 120, a command is driven onto an internal command bus 150 and placed in queues within the push/pull engine interface 122. Receive/read-related instructions (such as instructions for reading the CSRS) are written to a "push" command queue.

The CSRs 138 include the following types of registers: Fbus receive and transmit registers; Fbus and ready bus configuration registers; ready bus control registers; hash unit configuration registers; interrupt registers; and several miscellaneous registers, including a thread status registers. Those of the registers which pertain to the receive process will be described in further detail.

The interrupt/signal registers include an INTER_THD_SIG register for inter-thread signaling. Any thread within the microengines 22 or the core 23 can write a thread number to this register to signal an inter-thread event.

Further details of the Fbus controller 130 and the ready bus controller 126 are shown in FIG. 5. The ready

bus controller 126 includes a programmable sequencer 160 for retrieving MAC device status information from the MAC device 31 and asserting flow control to the MAC device over the ready bus 128 via ready bus interface logic 161. The Fbus controller 130 includes Fbus interface logic 162, which is used to transfer data to and from the device 31 is controlled by a transmit state machine (TSM) 164 and a receive state machine (RSM) 166. In the embodiment herein, the Fbus 132 may be configured as a bidirectional 64-bit bus, or two dedicated 32-bit buses. In the unidirectional, 32-bit configuration, each of the state machines owns its own 32-bit bus. In the bidirectional configuration, the ownership of the bus is established through arbitration. Accordingly, the Fbus controller 130 further includes a bus arbiter 168 for selecting which state machine owns the Fbus 132.

Some of the relevant CSRs used to program and control the ready bus 128 and Fbus 132 for receive processes are shown in FIGS. 6A-6F. Referring to FIG. 6A, RDYBUS_TEMPLATE_PROGx registers 170 are used to store instructions for the ready bus sequencer. Each register of these 32-bit registers 170a, 170b, 170c, includes four, 8-bit instruction fields 172. Referring to FIG. 6B, a

RCV_RDY_CTL register 174 specifies the behavior of the receive state machine 166. The format is as follows: a reserved field (bits 31:15) 174a; a fast port mode field (bits 14:13) 174b, which specifies the fast port thread mode, as will be described; an auto push prevent window field (bits 12:10) 174c for specifying the autopush prevent window used by the ready bus sequencer to prevent the receive scheduler from accessing its read transfer registers when an autopush operation (which pushes information to those registers) is about to begin; an autopush enable (bit 9) 174d, used to enable autopush of the receive ready flags; another reserved field (bit 8) 174e; an autopush destination field (bits 7:6) 174f for specifying an autopush operation's destination register; a signal thread enable field (bit 5) 174g which, when set, indicates the thread to be signaled after an autopush operation; and a receive scheduler thread ID (bits 4:0) 174h, which specifies the ID of the microengine thread that has been configured as a receive scheduler.

Referring to FIG. 6C, a REC_FASTPORT_CTL register 176 is relevant to receiving packet data from fast ports such as ports 33a and 33b. It enables receive threads to view the current assignment of header and body thread

assignments for these two fast ports, as will be described. It includes the following fields: a reserved field (bits 31:20) 176a; an FP2_HDR_THD_ID field (bits 19:15) 176b, which specifies the fast port 2 header receive (processing) thread ID; an FP2_BODY_THD_ID field (bits 14:10) 176c for specifying the fast port 2 body receive processing thread ID; an FP1_HDR_THD_ID field (bits 9:5) 176d for specifying the fast port 1 header receive processing thread ID; and an FP1_BODY_THD_ID field (bits 4:0) 176e for specifying the fast port 1 body processing thread ID. The manner in which these fields are used by the RSM 166 will be described in detail later.

Although not depicted in detail, other bus registers include the following: a RDYBUS_TEMPLATE_CTL register 178 (FIG. 6D), which maintains the control information for the ready bus and the Fbus controllers, for example, it enables the ready bus sequencer; a RDYBUS_SYNCH_COUNT_DEFAULT register 180 (FIG. 6E), which specifies the program cycle rate of the ready bus sequencer; and an FP_FASTPORT_CTL register 182 (FIG. 6F), which specifies how many Fbus clock cycles the RSM 166 must wait between the last data transfer and the next sampling of fast receive status, as will be described.

Referring to FIG. 7, the MAC device 31 provides transmit status flags 200 and receive status flags 202 that indicate whether the amount of data in an associated transmit FIFO 204 or receive FIFO 206 has reached a certain threshold level. The ready bus sequencer 160 periodically polls the ready flags (after selecting either the receive ready flags 202 or the transmit ready flags 200 via a flag select 208) and places them into appropriate ones of the CSRs 138 by transferring the flag data over ready bus data lines 209. In this embodiment, the ready bus includes 8 data lines for transferring flag data from each port to the Fbus interface unit 124. The CSRs in which the flag data are written are defined as RCV_RDY_HI/LO registers 210 for receive ready flags and XMIT_RDY_HI/LO registers 212 for transmit ready flags, if the ready bus sequencer 160 is programmed to execute receive and transmit ready flag read instructions, respectively.

When the ready bus sequencer is programmed with an appropriate instruction directing it to interrogate MAC receive ready flags, it reads the receive ready flags from the MAC device or devices specified in the instruction and places the flags into a RCV_RDY_HI register 210a and a RCV_RDY_LO register 210b, collectively, RCV_RDY registers

210. Each bit in these registers corresponds to a different device port on the I/O bus.

Also, and as shown in the figure, the bus interface unit 124 also supports two fast port receive ready flag pins FAST_RX1 214a and FAST_RX2 214b for the two fast ports of the fast MAC device 31. These fast port receive ready flag pins are read by the RSM 166 directly and placed into an RCV_RDY_CNT register 216. The RCV_RDY_CNT register 216 is one of several used by the receive scheduler thread to determine how to issue a receive request. It also indicates whether a flow control request is issued.

Referring to FIG. 8A, the format of the RCV_RDY_CNT register 216 is as follows: bits 31:28 are defined as a reserved field 216a; bit 27 is defined as a ready bus master field 216b and is used to indicate whether the ready bus 128 is configured as a master or slave; a field corresponding to bit 26 216c provides flow control information; bits 25 and 24 correspond to FRDY2 field 216d and FRDY1 field 216e, respectively. The FRDY2 216d and FRDY1 216e are used to store the values of the FAST_RX2 pin 214b and FAST_RX1 pin 214a, respectively, both of which are sampled by the RSM 166 each Fbus clock cycle; bits 23:16 correspond to a reserved field 216f; a receive request

count field (bits 15:8) 216g specifies a receive request count, which is incremented after the RSM 166 completes a receive request and data is available in the RFIFO 136; a receive ready count field (bits 7:0) 216h specifies a receive ready count, an 8-bit counter that is incremented each time the ready bus sequencer 160 writes the ready bus registers RCV_RDY_CNT register 216, the RCV_RDY_LO register 210b and RCV_RDY_HI register 210a to the receive scheduler read transfer registers.

There are two techniques for reading the ready bus registers: "autopush" and polling. The autopush instruction may be executed by the ready bus sequencer 160 during a receive process (rxautopush) or a transmit process (txautopush). Polling requires that a microengine thread periodically issue read references to the I/O bus interface 28.

The rxautopush operation performs several functions. It increments the receive ready count in the RCV_RDY_CNT register 216. If enabled by the RCV_RDY_CTL register 174, it automatically writes the RCV_RDY_CNT 216, the RCV_RDY_LO and RCV_RDY_HI registers 210b, 210a to the receive scheduler read transfer registers 80 (FIG. 2) and signals to the receive scheduler thread 92 (via a context event

signal) when the rxautopush operation is complete.

The ready bus sequencer 160 polls the MAC FIFO receive ready flags periodically and asynchronously to other events occurring in the processor 12. Ideally, the rate at which the MAC FIFO receive ready flags are polled is greater than the maximum rate at which the data is arriving at the MAC device ports. Thus, it is necessary for the receive scheduler thread 92 to determine whether the MAC FIFO receive ready flags read by the ready bus sequencer 160 are new, or whether they have been read already. The rxautopush instruction increments the receive ready count in the RCV_RDY_CNT register 216 each time the instruction executes. The RCV_RDY_CNT register 216 can be used by the receive scheduler thread 92 to determine whether the state of specific flags have to be evaluated or whether they can be ignored because receive requests have been issued and the port is currently being serviced. For example, if the FIFO threshold for a Gigabit Ethernet port is set so that the receive ready flags are asserted when 64 bytes of data are in the MAC receive FIFO 206, then the state of the flags does not change until the next 64 bytes arrive 5120 ns later. If the sequencer 160 is programmed to collect the flags four times each 5120 ns period, the

next three sets of ready flags that are collected by the ready bus sequencer 160 can be ignored.

When the receive ready count is used to monitor the freshness of the receive ready flags, there is a possibility that the receive ready flags will be ignored when they are providing new status. For a more accurate determination of ready flag freshness, the receive request count may be used. Each time a receive request is completed and the receive control information is pushed onto the RCV_CNTL register 232, the the RSM 166 increments the receive request count. The count is recorded in the RCV_RDY_CNT register the first time the ready bus sequencer executes an rxrdy instruction for each program loop. The receive scheduler thread 92 can use this count to track how many requests the receive state machine has completed. As the receive scheduler thread issues commands, it can maintain a list of the receive requests it submits and the ports associated with each such request.

Referring to FIGS. 8B and 8C, the registers RCV_RDY_HI 210a and RCV_RDY_LO 210b have a flag bit 217a, 217b, respectively, corresponding to each port.

Referring to FIG. 9, the receive scheduler thread 92 performs its tasks at a rate that ensures that the RSM 166

is always busy, that is, that there is always a receive request waiting to be processed by the RSM 166. Several tasks performed by the receive scheduler 92 are as follows. The receive scheduler 92 determines which ports need to be serviced by reading the RCV_RDY_HI, RCV_RDY_LO and RCV_RDY_CNT registers 210a, 210b and 216, respectively. The receive scheduler 92 also determines which receive ready flags are new and which are old using either the receive request count or the receive ready count in the RCV_RDY_CNT register, as described above. It tracks the thread processing status of the other microengine threads by reading thread done status CSRs 240. The receive scheduler thread 92 initiates transfers across the Fbus 132 via the ready bus, while the receive state machine 166 performs the actual read transfer on the Fbus 132. The receive scheduler 92 interfaces to the receive state machine 166 through two FBI CSRs 138: an RCV_REQ register 230 and an RCV_CNTL register 232. The RCV_REQ register 230 instructs the receive state machine on how to receive data from the Fbus 132.

Still referring to FIG. 9, a process of initiating an Fbus receive transfer is shown. Having received ready status information from the RCV_RDY_HI/LO registers 210a,

210b as well as thread availability from the thread done register 240 (transaction 1, as indicated by the arrow labeled "1"), the receive scheduler thread 92 determines if there is room in the RCV_REQ FIFO 230 for another receive request. If it determines that RCV_REQ FIFO 230 has room to receive a request, the receive scheduler thread 92 writes a receive request by pushing data into the RCV_REQ FIFO 230 (transaction 2). The RSM 166 processes the request in the RCV_REQ FIFO 230 (transaction 3). The RSM 166 responds to the request by moving the requested data into the RFIFO 136 (transaction 4), writing associated control information to the RCV_CTL FIFO 232 (transaction 5) and generating a start_receive signal event to the receive processing thread 96 specified in the receive request (transaction 6). The RFIFO 136 includes 16 elements 241, each element for storing a 64 byte unit or segment of data referred to herein as a MAC packet ("MPKT"). The RSM 166 reads packets from the MAC ports in fragments equal in size to one or two RFIFO elements, that is, MPKTs. The specified receive processing thread 96 responds to the signal event by reading the control information from the RCV_CTL register 232 (transaction 7). It uses the control information to determine, among other pieces of

information, where the data is located in the RFIFO 136. The receive processing thread 96 reads the data from the RFIFO 136 on quadword boundaries into its read transfer registers or moves the data directly into the SDRAM (transaction 8).

The RCV_REQ register 230 is used to initiate a receive transfer on the Fbus and is mapped to a two-entry FIFO that is written by the microengines. The I/O bus interface 28 provides signals (not shown) to the receive scheduler thread indicating that the RCV_REQ FIFO 236 has room available for another receive request and that the last issued request has been stored in the RCV_REQ register 230.

Referring to FIG. 10A, the RCV_REQ FIFO 230 includes two entries 231. The format of each entry 231 is as follows. The first two bits correspond to a reserved field 230a. Bit 29 is an FA field 230b for specifying the maximum number of Fbus accesses to be performed for this request. A THSG field (bits 28:27) 230c is a two-bit thread message field that allows the scheduler thread to pass a message to the assigned receive thread through the ready state machine, which copies this message to the RCV_CNTL register. An SL field 230d (bit 26) is used in

cases where status information is transferred following the EOP MPKT. It indicates whether two or one 32-bit bus accesses are required in a 32-bit Fbus configuration. An E1 field 230e (bits 21:18) and an E2 field (bits 25:22) 230f specify the RFIFO element to receive the transferred data. If only 1 MPKT is received, it is placed in the element indicated by the E1 field. If two MPKTs are received, then the second MPKT is placed in the RFIFO element indicated by the E2 field. An FS field (bits 17:16) 230g specifies use of a fast or slow port mode, that is, whether the request is directed to a fast or slow port. The fast port mode setting signifies to the RSM that a sequence number is to be associated with the request and that it will be handling speculative requests, which will be discussed in further detail later. An NFE field (bit 15) 230h specifies the number of RFIFO elements to be filled (i.e., one or two elements). The IGFR field (bit 13) 230i is used only if fast port mode is selected and indicates to the RSM that it should process the request regardless of the status of the fast ready flag pins. An SIGRS field (bit 11) 230j, if set, indicates that the receive scheduler be signaled upon completion of the receive request. A TID field (bits 10:6) 230k specifies

the receive thread to be notified or signaled after the receive request is processed. Therefore, if bit 11 is set, the RCV_REQ entry must be read twice, once by the receive thread and once by the receive scheduler thread, before it can be removed from the RCV_REQ FIFO. An RM field (bits 5:3) 230l specified the ID of the MAC device that has been selected by the receive scheduler. Lastly, an RP field (bits 2:0) 230m specifies which port of the MAC device specified in the RM field 230l has been selected.

The RSM 166 reads the RCV_REQ register entry 231 to determine how it should receive data from the Fbus 132, that is, how the signaling should be performed on the Fbus, where the data should be placed in the RFIFO and which microengine thread should be signaled once the data is received. The RSM 166 looks for a valid receive request in the RCV_REQ FIFO 230. It selects the MAC device identified in the RM field and selects the specified port within the MAC by asserting the appropriate control signals. It then begins receiving data from the MAC device on the Fbus data lines. The receive state machine always attempts to read either eight or nine quadwords of data from the MAC device on the Fbus as specified in the receive request. If the MAC device asserts the EOP signal, the RSM 166 terminates

the receive early (before eight or nine accesses are made). The RSM 166 calculates the total bytes received for each receive request and reports the value in the RCV_CNTL register 232. If EOP is received, the RSM 166 determines the number of valid bytes in the last received data cycle.

The RCV_CNTL register 232 is mapped to a four-entry FIFO (referred to herein as RCV_CNTL_FIFO 232) that is written by the receive state machine and read by the microengine thread. The I/O bus interface 28 signals the assigned thread when a valid entry reaches the top of the RCV_CNTL FIFO. When a microengine thread reads the RCV_CNTL register, the data is popped off the FIFO. If the SIGRS field 230i is set in the RCV_REQ register 230, the receive scheduler thread 92 specified in the RCV_CNTL register 232 is signaled in addition to the thread specified in TID field 230k. In this case, the data in the RCV_CNTL register 232 is read twice before the receive request data is retired from the RCV_CNTL FIFO 232 and the next thread is signaled. The receive state machine writes to the RCV_CNTL register 232 as long as the FIFO is not full. If the RCV_CNTL FIFO 232 is full, the receive state machine stalls and stops accepting any more receive requests.

Referring to FIG. 10B, the RCV_CNTL FIFO 232 provides instruction to the signaled thread (i.e., the thread specified in TID) to process the data. As indicated above, the RCV_CNTL FIFO includes 4 entries 233. The format of the RCV_CNTL FIFO entry 233 is as follows: a THMSG field (31:30) 23a includes the 2-bit message copied by the RSM from RCV_REQ register[28:27]. A MACPORT/THD field (bits 29:24) 232b specifies either the MAC port number or a receive thread ID, as will be described in further detail below. An SOP SEQ field (23:20) 232c is used for fast ports and indicates a packet sequence number as an SOP (start-of-packet) sequence number if the SOP was asserted during the receive data transfer and indicates an MPKT sequence number if SOP was not so asserted. An RF field 232d and RERR field 232e (bits 19 and 18, respectively) both convey receive error information. An SE field 232f (17:14) and an FE field 232g (13:10) are copies of the E2 and E1 fields, respectively, of the RCV_REQ. An EF field (bit 9) 232h specifies the number of RFIFO elements which were filled by the receive request. An SN field (bit 8) 232i is used for fast ports and indicates whether the sequence number specified in SOP_SEQ field 232c is associated with fast port 1 or fast port 2. A VLD BYTES

field (7:2) 232j specifies the number of valid bytes in the RFIFO element if the element contains in EOP MPKT. An EOP field (bit 1) 232k indicates that the MPKT is an EOP MPKT. An SOP field (bit 0) 232l indicates that the MPKT is an SOP MPKT.

The thread done registers 240 can be read and written to by the threads using a CSR instruction. Using these registers, the receive scheduler thread can determine which RFIFO elements are not in use. The THREAD_DONE CSRs 240 support a two-bit message for each microengine thread. The assigned receive thread may write a two-bit message to this register to indicate that it has completed its task. Each time a message is written to the THREAD_DONE register, the current message is logically ORed with the new message. The bit values in the THREAD_DONE registers are cleared by writing a "1", so the scheduler may clear the messages by writing the data read back to the THREAD_DONE register. The definition of the 2-bit status field is determined in software.

The assigned receive processing threads write their status to the THREAD_DONE register whenever the status changes. When the receive scheduler reads the THREAD_DONE register, it can look at the returned value to determine

the status of each thread and then update its thread/port assignment list.

The packet rate of a fast port (e.g., a Gigabit port) is such that the rate at which the receive state machine reads MPKTs from a single port is so fast that a receive thread may not be able to process an MPKT before the receive state machine brings in another MPKT from the same port. That is, a fast port may require the use of a number of RFIFO elements and receive threads in parallel to maintain full line rate. The amount of processing required for an MPKT may include header processing (e.g., header modification, forward lookup) or simply moving a packet body fragment to memory.

Fast packets and, in some cases, fast MPKTs (i.e., MPKTs which make up packets received from fast ports) can be processed in parallel and by different threads, so there is a need to maintain intra-packet order and inter-packet order for a given port. Thus, to maintain packet order for packets received from fast ports, the network processor 12 uses sequence numbers, one set for each high-speed port. Each set of sequence numbers provides a network packet sequence number, an MPKT sequence number and an enqueue sequence number. These sequence numbers are maintained as

4-bit counters within the I/O bus interface 28 and automatically roll over to zero once they reach a count of fifteen.

The sequence numbers are maintained in Fbus receive registers (CSRs). Referring to FIG. 11A, sequence numbers registers 270 include an SOP_SEQ1 register 272 having an SOP_SEQ1 field 273 and an SOP_SEQ2 register 274, which has an SOP_SEQ2 field 275. These fields store SOP sequence numbers for their respective fast ports and are incremented by the RSM. Referring to FIG. 11B, enqueue sequence number registers 276 include an ENQUEUE_SEQ1 register 278 having an ENQUEUE_SEQ1 field 279 for storing an enquence sequence number for fast port 1 and an ENQUEUE_SEQ2 register 280, which includes an ENQUEUE_SEQ2 field 281 for storing enqueue SOP sequence number for fast port 2. The enqueue sequence numbers are incremented by the receive processing threads.

The network packet sequence number in either the SOP_SEQ1 register (for fast port 1) or SOP_SEQ2 register (for fast port 2) register is placed into the RCV_CNTL register, and incremented at the same time. The receive state machine increments the packet sequence numbers in a manner that allows the receive processing threads to track

not only the sequence of the network packets, but also the sequence of the individual MPKTs. If the SOP signal is detected during a receive request, the network packet sequence number provides a sequence number based on a network packet (hereinafter referred to as an SOP sequence number). If the SOP signal is not detected during a receive request, the packet sequence number is based on an MPKT (hereinafter, MPKT sequence number). The receive threads can determine the type of packet sequence number since the RCV_CNTL register contains both the packet sequence number and SOP status.

The SOP and MPKT sequence numbers for each fast port are implemented as 4-bit counters. The SOP sequence number counter is incremented each time an SOP is detected. An MPKT sequence number counter receives the SOP sequence number whenever the SOP signal is asserted, and is incremented once per receive request when the SOP signal is not detected.

The enqueue sequence numbers are used by the receive processing threads to determine whether it is their turn to place a complete network packet onto a transmit queue. When an entire network packet has been received, the receive processing thread reads the enqueue sequence number

from the appropriate enqueue_seq register. If the enqueue sequence number matches the SOP sequence number assigned to the packet, the receive processing thread can place the packet onto a transmit queue. If the enqueue sequence number does not match, the receive processing thread waits for a "sequence number change" signal event to occur. When the event occurs, the receive processing thread reads the enqueue sequence number again and checks for a match. If a match occurs, the packet may be placed onto a transmit queue.

After a packet is placed on a transmit queue, the receive processing thread increments the enqueue sequence number. The enqueue sequence numbers are incremented by writing to either the ENQUEUE_SEQ1 or ENQUEUE_SEQ2 register. A receive processing thread may choose to write its processing status to the THREAD_DONE register as well as increment the enqueue sequence number at the same time. This can be accomplished with a single write instruction to additional CSRs, a THREAD_DONE_INCR1 register or the THREAD_DONE_INCR2 register (not shown).

The receive scheduler thread controls the rate at which it issues receive requests. It issues a number of receive requests that is no more than that required by a

port, but is sufficient to prevent an overflow of that port's receive FIFO.

When using slower ports, such as 10/100 BaseT Ethernet ports, the receive scheduler thread reads the MAC receive FIFO ready flags for multiple ports, determines which ports have data available, and issues receive requests based on the knowledge that data is available in the MAC receive FIFO. Since it reads multiple receive FIFO ready flags each time, it can issue multiple receive requests before it has to read the flags again. Because fast ports operate at a much higher data rate than slow ports and the latencies associated with certain tasks, e.g., reading the receive ready flags from a port or from the RCV_RDY_HI/LO registers, writing a receive request to RCV_REQ, may be greater than that packet arrival rate, the rate at which a single MAC port must be serviced cannot be sustained by issuing receive requests only when data is known to be available in a device port receive FIFO.

Therefore, the receive scheduler thread uses speculative requests for high-speed ports. That is, the receive scheduler thread issues multiple receive requests to a port based on the speculation that there is data available in that port's receive FIFO. At the time the RSM

166 processes each receive request, it determines if data is actually available at the port. Based on this determination, the RSM 166 either processes or cancels the request.

The RSM 166 determines whether there is data available at either of the two fast ports by reading the fast receive ready pins (FAST_RX1 214a and FAST_RX2 214b of FIG. 7). These pins 214a, 214b provide a direct connection to their respective MAC port's receive FIFO ready flag. The MAC ports assert these signals when the receive FIFO fullness threshold level is reached or an entire packet has been received.

If a fast ready pin in not asserted, the RSM 166 cancels the pending request and writes a cancel message into the RCV_CNTL register's message field. It then signals the assigned receive processing thread. The receive processing thread is programmed to read the RCV_CNTL register, interpret the cancel message correctly and indicate to the receive scheduler thread that it is available for other tasks.

The state of the two fast ready pins is indicated in the FRDY2 field 216d (for port 2) and FRDY1 field 216e (for port 1) of the RCV_RDY_CNT register 216 (shown in FIG. 8A).

The receive scheduler thread reads the fast ready flags from the RCV_RDY_CNT register 216 fields 216d, 216e on a periodic basis to determine when it should issue receive requests. It issues enough receive requests to cover the data that might have arrived in the MAC port 33 since the last time it read the fast ready flags.

The receive state machine 166 supports three fast port modes that determine how receive processing threads are assigned to process packet data in the RFIFO. These fast port modes are referred to as single thread, header/body thread and explicit thread modes. When selecting a mode, the network processor considers the following: availability of threads to process each receive request; execution time for the receive thread. The modes need to understand where one network packet ends and the next one begins. To that end, they rely on the beginning of the network packet as corresponding to the assertion of SOP and the ending of the network packet corresponding to the assertion of EOP. Referring back to FIG. 6B, the fast port mode field 174b of RCV_RDY_CTL register 176 defines the three modes as single thread '00', header/body '01' and explicit '10'.

The single thread mode assigns a single thread to

each packet when using speculative requests. If the single thread mode is specified in the RCV_RDY_CTL register 176 and fast port thread mode (RCV_REQ[17:16]) is set, the RSM 166 performs in the following manner. If the RSM 166 detects an SOP in the receive data transfer for the MPKT, it signals the thread specified in the RCV_REQ register 230. That is, it writes the thread ID of the specified thread to the TID field 230k. It also saves that thread ID in the appropriate header field of the REC_FASTPORT_CTL register 176. If SOP is not detected, the RSM 166 ignores the thread ID specified in the RCV_REQ register and signals the thread specified in the header field in the REC_FASTPORT_CTL register. The RSM 166 writes the unused thread ID to the RCV_CNTL register MACPORT/THD field 232b. The unused ID is returned to the receive scheduler thread so the receive scheduler thread can update its thread availability list. To return the thread ID, the RSM 166 signals the receive thread when the receive request is complete and the receive thread passes the unused thread ID to the receive scheduler using inter-thread communications. Alternatively, the receive scheduler thread can request that it be signaled as well as the receive processing thread after the RSM completes the receive request. In

this case, RCV_CNTL must be read twice before data is removed from the RCV_CNTL FIFO. In most cases, the receive processing thread reads it once and the receive scheduler thread also reads it once. If two reads are not performed, the RSM stalls. In another alternative, the RSM signals the receive processing thread when the receive request is complete and the receive processing thread returns the unused thread to the receive scheduler thread using an inter-thread signaling register which, like the thread done registers, has a bit for each thread and is read periodically by the receive scheduler to determine thread availability. It sets the bit corresponding to the unused thread ID in that register, which is then read by the receive scheduler thread.

In the header/body mode, two threads are assigned to process the MPKTs within a network packet. The first thread serves as the header thread and is responsible for processing the header to determine how to forward the packet. The second thread is the body thread, which is responsible for moving the remainder of the packet to the SDRAM. When the body thread completes its task, it uses inter-thread signaling to notify the header thread where the body of the packet is located. The header thread can

then place the packet onto a transmit queue.

The RSM 166 supports the header and body threads in the following manner. If the RSM 166 detects an SOP, it signals the thread specified in RCV_REQ register and saves the thread number in the header field of REC_FASTPORT_CTL register 176. When it processes the next request, it signals the thread specified in RCV_REQ register 230 and saves the thread number in the body field of REC_FAST_PORT_CTL register 176. From this point forward, the RSM ignores the thread ID presented in the RCV_REQ register 230 and signals the body thread specified in REC_FASTPORT_CTL register 176. The RSM writes the unused thread ID to the RCV_CNTL register's MACPORT/THD field 232b. As with the single thread mode, the unused thread ID is returned to the receive scheduler thread so the receive scheduler thread knows that the thread is available for processing.

In explicit thread mode, the RSM always uses the thread assignment in the receive request as indicated by the RCV_REQ register 230. In this mode, the receive scheduler thread provides each receive processing thread with the ID of the thread assigned to the next MPKT receive request so that the thread can signal the next assigned

thread for the next consecutive MPKT that is it done, the exception being the last thread in line, which receives instead the thread ID of the header thread. Additionally, each thread provides the next assigned thread with a pointer to the buffer memory, thus ensuring the MPKTs for a given network packet are queued in packet memory in the order in which they arrived. Once the thread assigned to the EOP MPKT has completed processing and has been signaled by the thread for the previous MPKT, it notifies the header thread that the entire packet can be enqueued on the transmit queue, provided, that is, that the enqueue sequence number matches the SOP sequence number of the MPKT processed by the header thread. The MPKT sequence number is provided to ensure that MPKTS are queued in the correct order.

Referring to FIG. 12, an overview of the fast port receive processing for a selected fast port 300 is shown. The receive scheduler thread selects or assigns 302 an available thread to the port and issues 304 a receive request specifying the assigned thread. As noted in dashed lines, in explicit mode, the scheduler selects 306 a secondary thread as a thread to be assigned in the next receive request and stores the secondary thread in a memory

location designated as corresponding to the RFIFO element to which it will be written. The RSM checks 308 the fast ready flag for the fast port. The RSM determines 310 if it is asserted. If it is asserted, the RSM processes 312 the receive request, and responds to the request by transferring 314 the requested MPKT into an RFIFO element indicated by the request, and performs the step of posting 316 a RVC_CNTL FIFO entry (according to the fast port mode specified in the RCV_RDY_CTL register 174) to the RCV_CNTL FIFO and, at the same time, signaling the assigned thread (and any other threads, e.g., the scheduler, as specified by the request). Once these steps are completed, the assigned receive processing thread processes 318 the MPKT as instructed by the control information in the RCV_CNTL register and the fast port mode. If the ready flag is not asserted, it determines 319 if the IGFR field is set in the RCV_REQ entry. If not set, the RSM cancels 320 the request and returns the ID of the thread. If it is set, the RSM proceeds to process the request.

Referring to FIG. 13A, the RCV_CNTL entry posting and thread signaling of 316 (FIG. 12) includes, for the single thread mode, the following. The RSM determines 330 if SOP is asserted during the receive data cycle. If so,

it places 332 the SOP sequence number in the SOP_SEQ field, increments 334 the SOP_SEQx counter, sets 336 the SOP bit, writes 338 the specified thread ID to the TID field as well as saves 340 it in the REC_FASTPORT_CTL register header field for the appropriate fast port. It signals 342 the specified thread. If SOP is not asserted, the RSM writes 346 the MPKT sequence number to the sequence number field, and increments 348 that number. It sets 350 the TID field to the ID of the thread indicated in the header field of the REC_FASTPORT_CTL register (i.e., the TID for the last MPKT that was an SOP MPKT). It also writes 352 the unused receive processing thread, that is, the thread specified by the receive request to the MACPORT/THD field. It signals 354 both the assigned thread and the scheduler to read the register, the assigned thread so that it knows how to process the packet and the receive scheduler thread so that it knows the specified thread was not used and is therefore available for a new assignment.

Referring to FIG. 13B, the processing of the MPKT (318, FIG. 12) for the single thread mode is as follows. If the assigned processing thread determines 360 that the MPKT is as SOP MPKT (as indicated by the RCV_CNTL register), the assigned processing thread parses 362 the

header and performs a lookup 364 (based in the header and hash information retrieved from the hash table). It moves 366 both the header as processed, along with forwarding information stored in the SDRAM forwarding tables and the remainder of the MPKT (i.e., payload) into a temporary queue in packet buffer memory. If it determines 368 that the MPKT is an EOP, then the assigned thread assumes 370 that the packet is ready to be enqueued in the transmit queue for the forwarding port indicated by the forwarding information. The enqueuing process will be described with reference to FIG. 18. If the MPKT is not an SOP, the processing thread moves 372 the payload data to buffer memory (in SDRAM) and then determines 374 if it is an EOP. If it is an EOP, the processing thread is ready to enqueue the packet 376. It the MPKT is not an EOP, then the processing thread signals 378 that it is done (via inter-signaling methods, e.g., write thread done register).

Referring to FIG. 14A, the RCV_CNTL entry posting and signaling of threads 316 includes, for the dual (or header/body) thread mode, the following. The RSM determines 380 if SOP is asserted during the receive data cycle. If so, it places 382 the SOP sequence number in the SOP_SEQx field, increments 384 the SOP_SEQx counter, writes

386 the specified thread ID to the TID field as well as saves 388 it in the REC_FASTPORT_CTL register header field for the appropriate fast port. It signals 390 the specified thread. If SOP is not asserted, the RSM writes 392 the MPKT sequence number to the sequence number field, and increments 394 that number. It determines 396 if the last request was for an SOP MPKT. If so, it signals 398 the specified thread, sets 400 the ID of that thread in the TID field as well as the appropriate body field of the REC_FASTPORT_CTL register. It also indicates 402 in the MACPORT/THD field the ID of the header thread (so that the header thread may be signaled when the entire packet has been received and processed). If the last request was not an SOP MPKT, the RSM signals 404 the thread specified in the body, writes 406 that ID to TID field and specifies 408 the ID of the unused thread of the receive request in the MACPORT/THD field (to return to the pool of available receive processing threads). It also signals 410 the scheduler so that the scheduler, in addition to the signaled receive processing thread, may read the rec_cntl register entry before it is removed from the RCV_CNTL FIFO.

Referring to FIG. 14B, the MPKT is processed by the "assigned" thread in the dual thread mode as follows. If

the thread determines 412 that the MPKT is an SOP MPKT, it processes 414 the header and payload data in the same manner as described above (i.e., parses the header, etc.). If it determines 416 that the MPKT being processing is an EOP, that is, the MPKT is a minimum sized network packet, then it assumes 418 the MPKT is ready for enqueuing. If the MPKT is not the last MPKT in a packet, then the thread (which is the header thread) awaits notification 420 of EOP. Once it receives such notification 422, the packet is ready to be enqueued in the transmit queue. If the MPKT is not an SOP but the continuation of a packet, the thread stores 424 the payload in the temporary queue in SDRAM at a buffer location designated by the header thread. If it determines 426 that the MPKT is an EOP, then it signals 428 to the scheduler and the header thread (as identified in the MACPORT/THD field) that it is done. It thus determines 430 that the complete packet is now ready to be enqueued. If the MPKT is not an EOP, it simply signals 432 to the scheduler that it is done processing its MPKT and is available for work.

Referring to FIG. 15A, the posting of the RCV_CNTL entry and signaling of threads includes, for the explicit mode, the following steps. As in the other fast

port modes, the RSM determines 440 if SOP is asserted during the receive data cycle. If so, it places 442 the SOP sequence number in the SOP_SEQx field, increments 444 the SOP_SEQx counter, writes 446 the specified thread ID to the TID field. It signals 448 the specified thread. If SOP is not asserted, the RSM writes 450 the MPKT sequence number to the sequence number field, increments 452 that number and signals 454 the specified thread.

Referring to FIG. 15B, the receive thread processing the fast port MPKT according to the explicit mode as follows. If the specified thread determines 460 that the MPKT is an SOP MPKT, the specified thread processes 462 the header, moves the payload and processed header to buffer memory 464. If it determines 465 that an EOP bit is set in the RCV_CNTL register entry, then it concludes 466 that the MPKT is ready to be enqueued in the appropriate port transmit queue. If the EOP is not set, that is, the MPKT is not an EOP MPKT, the thread (in this case, the header thread) passes 468 a pointer to the next available buffer location to the secondary thread ID that was specified by the scheduler in a common location corresponding to the RFIFO element in which the MPKT was stored. It then awaits

notification 470 from the EOP thread. If the MPKT is not an SOP MPKT, it receives 472 a pointer to a buffer location in SDRAM and queues 474 the MPKT in the buffer memory at the location pointed to by the pointer. If the thread determines 475 that the MPKT is an EOP MPKT, the thread signals 476 that it is done and that the MPKT is an EOP so that the header thread know that the network packet to which this EOP MPKT belongs is ready to be enqueued in the transmit queue. If the MPKT is not an EOP, the processing thread increments 478 the pointer to the next available buffer location and passes 480 the pointer to the thread processing the next, consecutive MPKT, that is, the ID specified by the scheduler as the secondary thread in a memory location corresponding to the RFIFO element in which the MPKT was stored.

Referring to FIG. 16, the process of enqueuing is illustrated. The header thread (which has identified an EOP or received notification of an EOP from another thread, as previously described), first determines if it is this particular packet's turn to be enqueued. It determines 490 if the enqueue sequence # is equal to the SOP sequence number that was associated with the SOP MPKT. If they are equal, the header thread links 494 the network packet (now

stored in its entirety in the packet buffer memory in SDRAM) to the port transmit queue (located in SRAM). It increments 496 the enqueue sequence number and notifies 498 the scheduler of completion. If the SOP sequence number is not equal to the enqueue sequence number, it waits to receive a signal indicating that the SOP sequence number has changed 500 and again compares the two sequence numbers.

It will be appreciated that the processes depicted in FIGS. 12-16 assume that no packet exemptions occurred, that the thread are able to handle the packet processing without assistance from the core processor. Such assistance, if invoked, in no way changes the manner in which packet order is maintained. Further, the processes of FIGS. 12-16 assume the availability of FIFO, e.g., RFIFO, space. Although not described in the steps of FIGS. 12-16 above, it will be appreciated that the various state machines must determine if there is room available in a FIFO prior to writing new entries to that FIFO. If a particular FIFO is full, the state machine will wait until the appropriate number of entries has been retired from that FIFO.

Additions, subtractions, and other modifications of

the preferred embodiments of the invention will be apparent to those practiced in this field and are within the scope of the following claims.

What is claimed is:

1. A method of forwarding data, comprising:

associating control information with data received from a first port; and

using the associated control information to enqueue the data for transmission to a second port in the same order in which the data was received from the first port.

2. The method of claim 1, wherein the control information includes sequence numbers.

3. The method of claim 2, wherein the data comprises units of data and the units of data are associated with a network packet.

4. The method of claim 3, wherein associating associates first sequence numbers with the units of data as they are received from the first port.

5. The method of claim 1, further comprising:

processing the units of data by receive processing

program threads to determine the second port to which the units of data are to be transmitted.

6. The method of claim 4, wherein associating further comprises maintaining second sequence numbers, and wherein using comprises:

determining if the first sequence numbers are equal to the second sequence numbers to order the units of data after the units of data are processed.

7. The method of claim 2, further comprising:

controlling transfer of the data from the first port to the receive processing program threads for processing.

8. The method of claim 7, wherein controlling comprises:

assigning the receive processing program threads to process the data.

9. The method of claim 8, wherein controlling comprises:

directing the units of data to the attention of the respective assigned receive processing program threads.

10. The method of claim 8, wherein controlling comprises:

directing the units of data to the attention of a single one of the respective assigned receive processing program threads.

11. The method of claim 8, wherein controlling comprises:

directing the units of data for processing by a first and a second different receive processing program thread, the first processing program thread handling a first one of the data units that comprises a packet header and a portion of payload data, and the second receive processing program thread handling a second one of the data units that comprises another portion of the payload data.

12. The method of claim 3, further comprising:

maintaining a sequence number count for generating start-of-packet sequence numbers; and

incrementing the sequence number count upon associating a current count value of the sequence number count with a data unit as an start-of-packet sequence

number when the data unit is recognized as corresponding to a start of a new packet.

13. The method of claim 3, further comprising:

maintaining an enqueue sequence number count for generating enqueue sequence numbers; and

incrementing the enqueue sequence number count after determining that a packet is ready to be enqueued for tranmission to the second port.

14. A processor for forwarding data from a first port to a second port comprises:

a microengine for executing program threads, the threads including a receive scheduler program thread for issuing requests for transfer of units of data from the first port and receive processing program threads;

a bus interface, responsive to the microengine, for receiving the units of data from the first port and directing the units of data to the receive processing program threads for processing and enqueueing the units of data in the order in which they were received from the first port for transmission to the second port.

15. The processor of claim 14, wherein the bus interface uses sequence numbers to ensure that the units of data are enqueued in the order in which they were received from the first.

16. The processor of claim 15, wherein the bus interface associates a first set of the sequence numbers with the units of data as they are received from the first port and maintains a second set of sequence numbers for use by the receive processing program threads in determining the order in which the units of data are to be enqueued.

17. The processor of claim 14, wherein the bus interface indicates to the receive scheduler program thread whether the first port has data available for processing by one or more of the receive processing program threads.

18. The processor of claim 14, wherein the receive scheduler program thread assigns available threads from among the one or more receive processing program threads to process the units of data.

19. The processor of claim 14, wherein the bus interface

comprises a receive state machine for controlling the transfer of the data units from the first port.

20. The processor of claim 19, wherein the units of data are associated with a network packet.

21. The processor of claim 20, wherein the receive scheduler program thread assigns each of the units of data to different ones of the receive processing program threads.

22. The processor of claim 21, wherein the receive state machine directs the units of data to the attention of the respective assigned different ones of the receive processing program threads.

23. The processor of claim 21, wherein the receive state machine directs the units of data to the attention of a single one of the respective assigned different ones of the receive processing program threads.

24. The processor of claim 21, wherein the receive state machine directs the units of data for processing by a first

and a second different receive processing program thread, the first processing program thread handling a first one of the data units that comprises a packet header and a portion of payload data, and the second receive processing program thread handling a second one of the data units that comprises another portion of the payload data.

25. An article comprising a computer-readable medium which stores computer-executable instructions for forwarding data, the instructions causing a computer to:

associate control information with data received from a first port; and

use the associated control information to enqueue the data for transmission to a second port in the same order in which the data was received from the first port.

26. The article of claim 25, wherein the control information includes sequence numbers.

27. The article of claim 26, wherein the instructions to use the associated control information comprise instructions causing a computer to:

determine the order in which the data are to be enqueued from the sequence numbers.

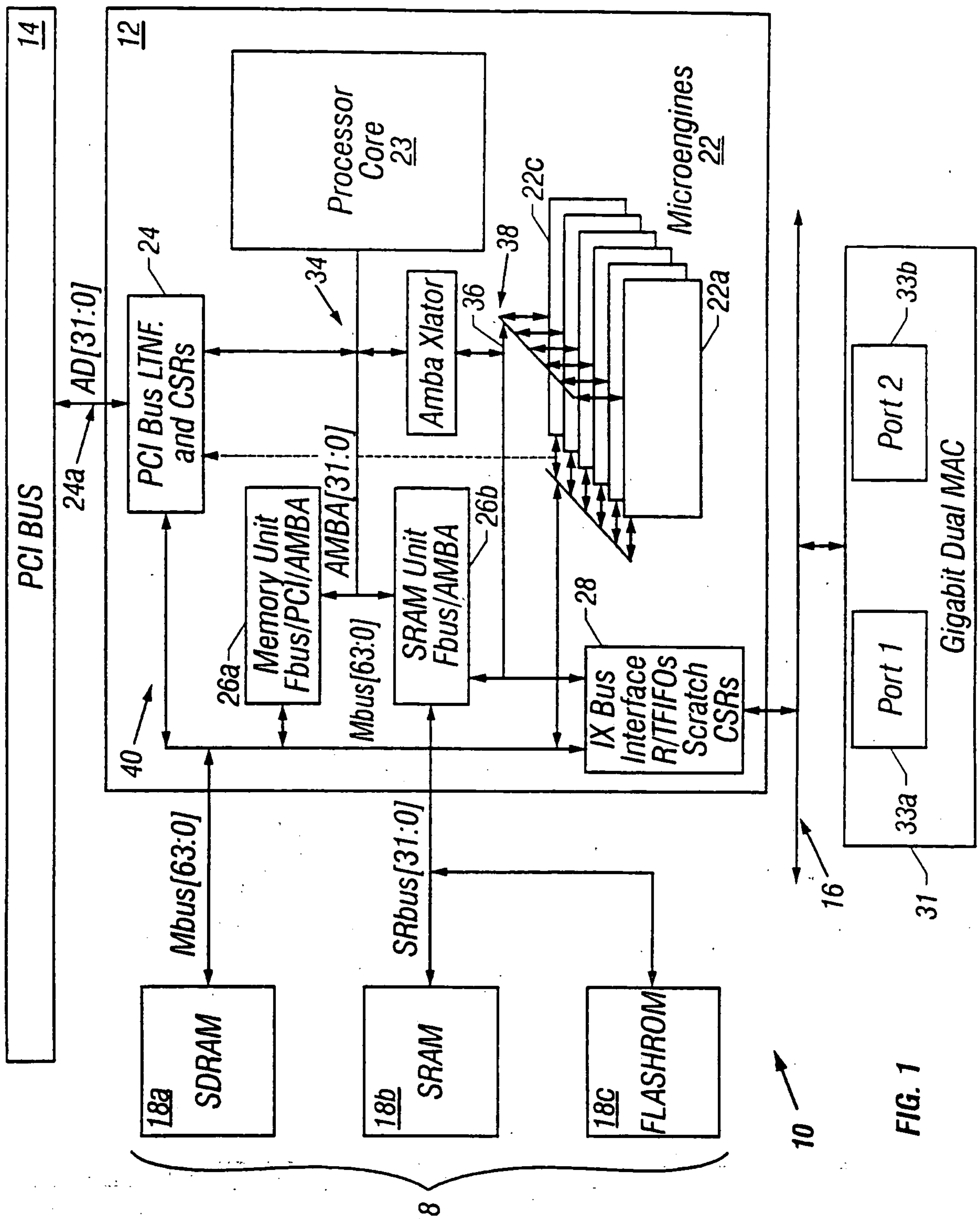

PCI BUS
14
12
Processor
Core
23
Microengines
22
22c
22a
38
36
34
24
AD[31:0]
24a
PCI Bus LTNF.
and CSRs
Amba Xlator
AMBA[31:0]
Memory Unit
Fbus/PCI/AMBA
26a
SRAM Unit
Fbus/AMBA
26b
Mbus[63:0]
28
IX Bus
Interface
R/TFIFOs
Scratch
CSRs
40
Port 2
33b
Port 1
33a
Gigabit Dual MAC
31
16
Mbus[63:0]
SRbus[31:0]
18a
SDRAM
18b
SRAM
18c
FLASHROM
18
10

FIG. 1

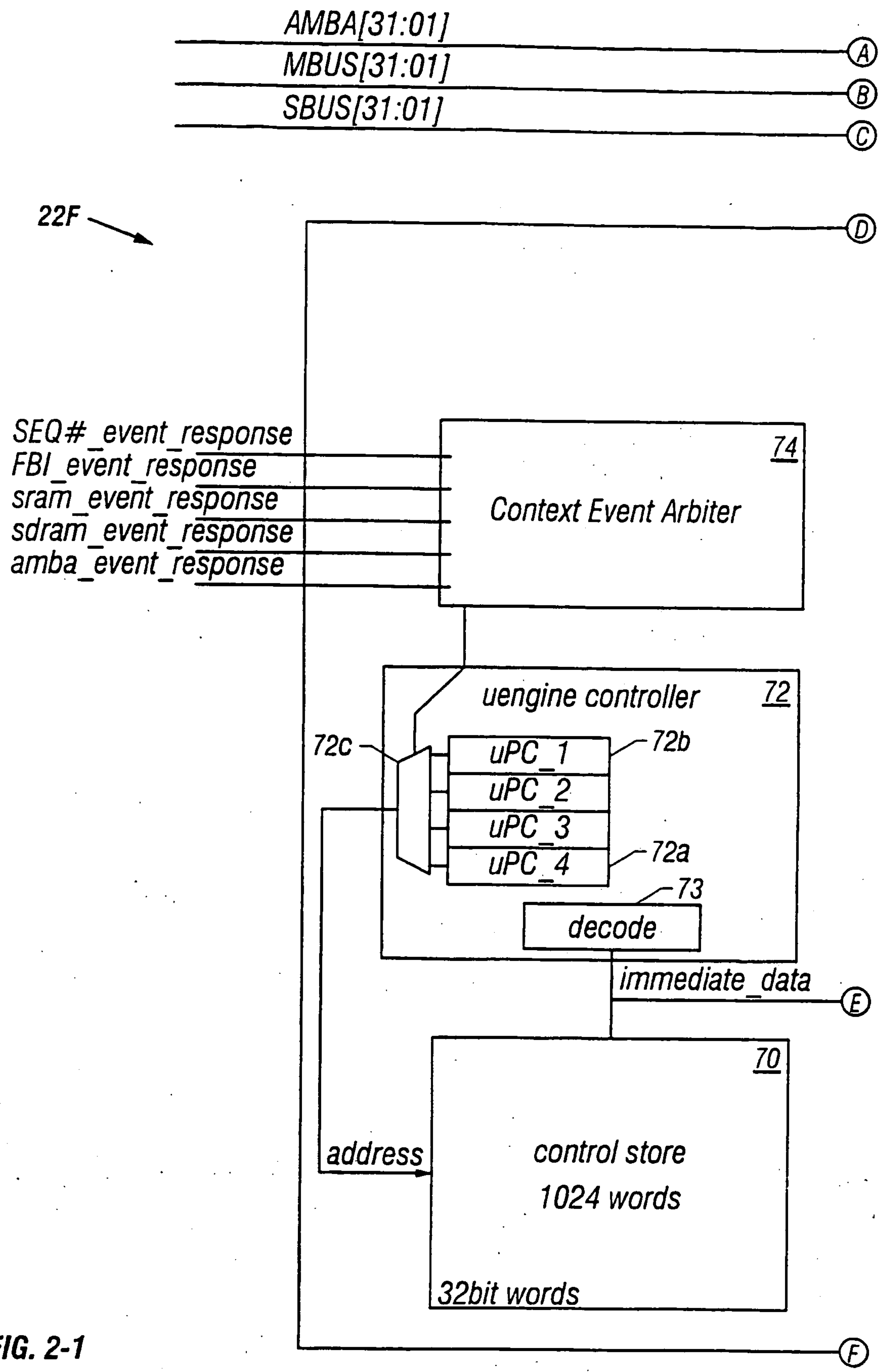

AMBA[31:01]
MBUS[31:01]
SBUS[31:01]
A
B
C
D
22F
SEQ#_event_response
FBI_event_response
sram_event_response
sdram_event_response
amba_event_response
Context Event Arbiter
74
uengine controller
72
72c
uPC_1
uPC_2
uPC_3
uPC_4
72b
72a
73
decode
immediate_data
E
70
control store
1024 words
32bit words
address
F

FIG. 2-1

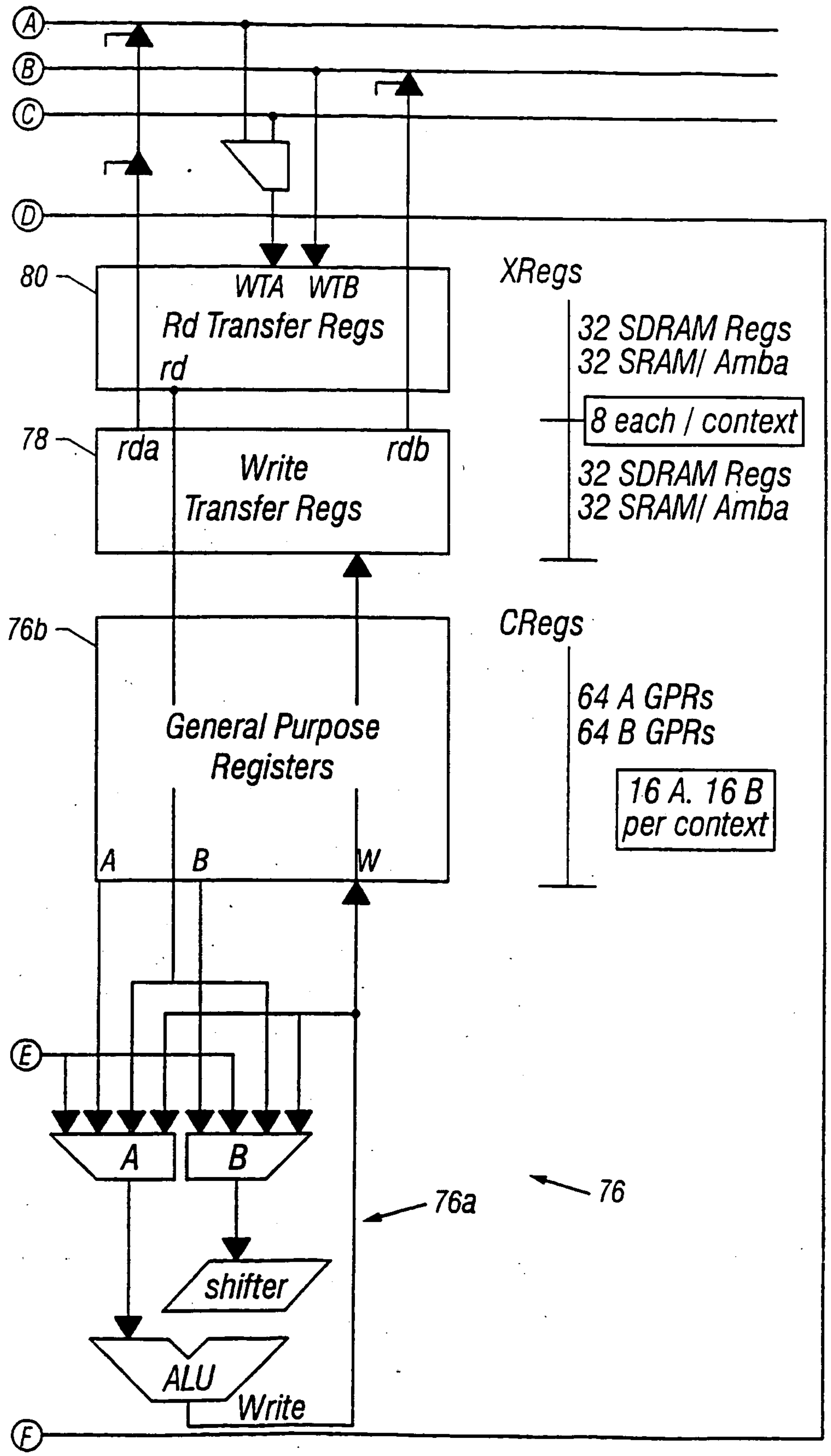

A
B
C
D
80
WTA WTB
Rd Transfer Regs
rd
XRegs
32 SDRAM Regs
32 SRAM/ Amba
8 each / context
78
rda
Write
Transfer Regs
rdb
32 SDRAM Regs
32 SRAM/ Amba
76b
General Purpose
Registers
CRegs
64 A GPRs
64 B GPRs
16 A. 16 B
per context
A
B
W
E
A
B
76a
76
shifter
ALU
Write
F

FIG. 2-2

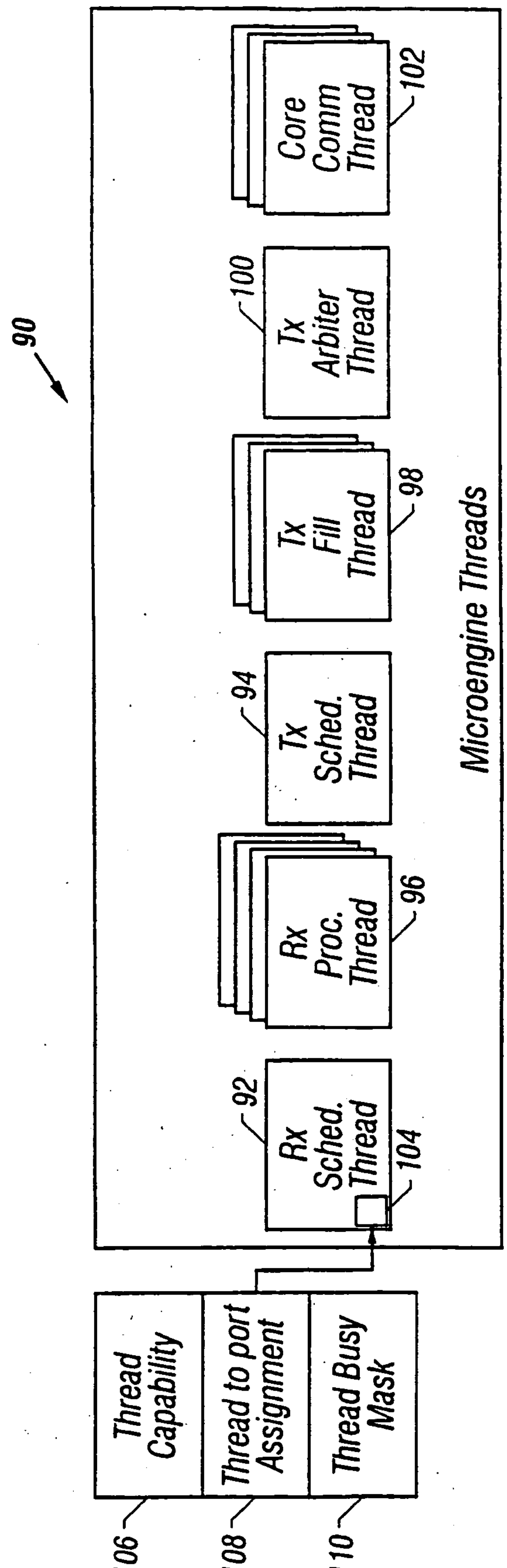

90
Microengine Threads
Rx Sched. Thread
92
104
Rx Proc. Thread
96
Tx Sched. Thread
94
Tx Fill Thread
98
Tx Arbiter Thread
100
Core Comm Thread
102
Thread Capability
106
Thread to port Assignment
108
Thread Busy Mask
110

FIG. 3

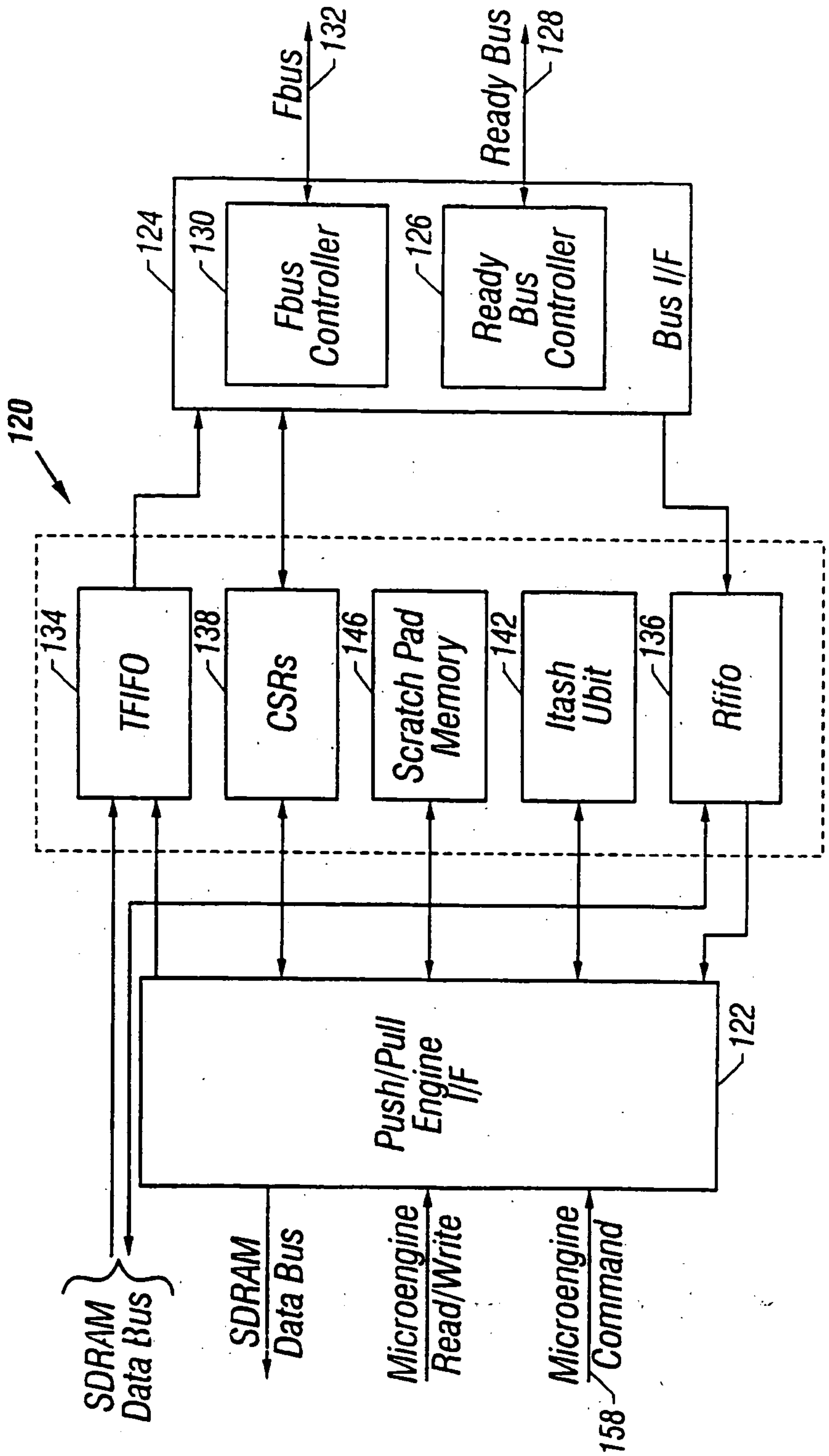

Fbus
132
Ready Bus
128
124
130
Fbus Controller
126
Ready Bus Controller
Bus I/F
120
134
TFIFO
138
CSRs
146
Scratch Pad Memory
142
Itash Ubit
136
Rfifo
Push/Pull Engine I/F
122
SDRAM Data Bus
SDRAM Data Bus
Microengine Read/Write
Microengine Command
158

FIG. 4

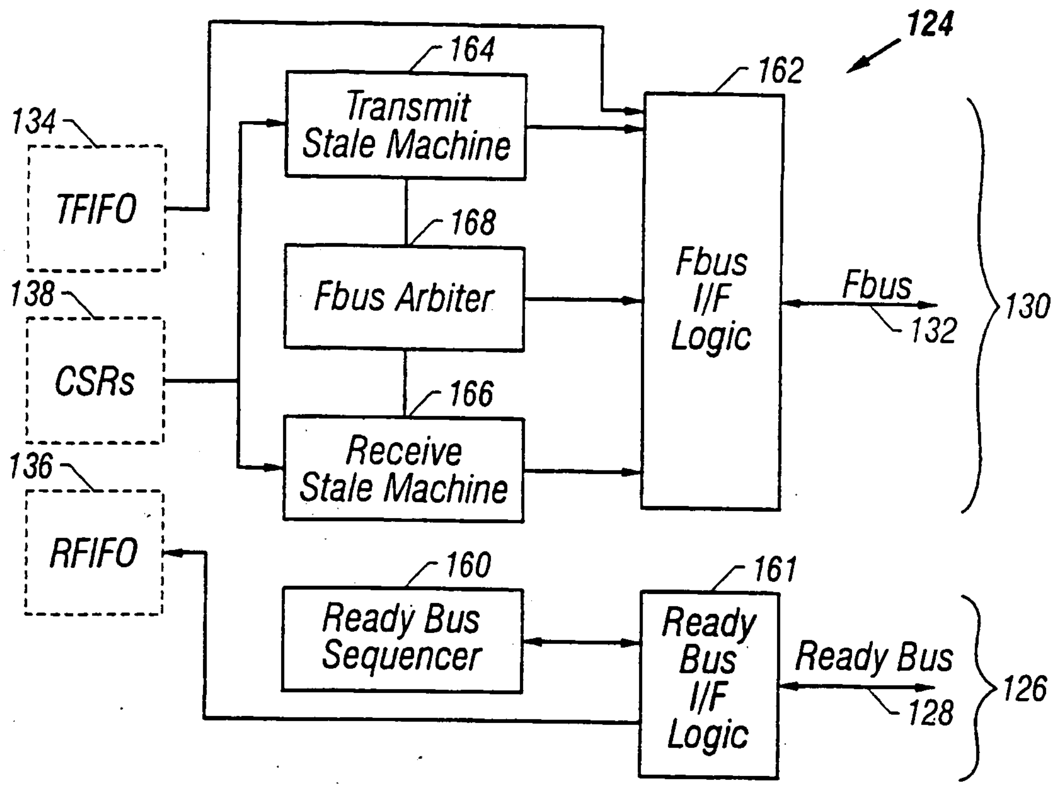

124
164
Transmit Stale Machine
162
134
TFIFO
168
Fbus Arbiter
Fbus I/F Logic
Fbus
132
130
138
CSRs
166
136
Receive Stale Machine
RFIFO
160
161
Ready Bus Sequencer
Ready Bus I/F Logic
Ready Bus
128
126

FIG. 5

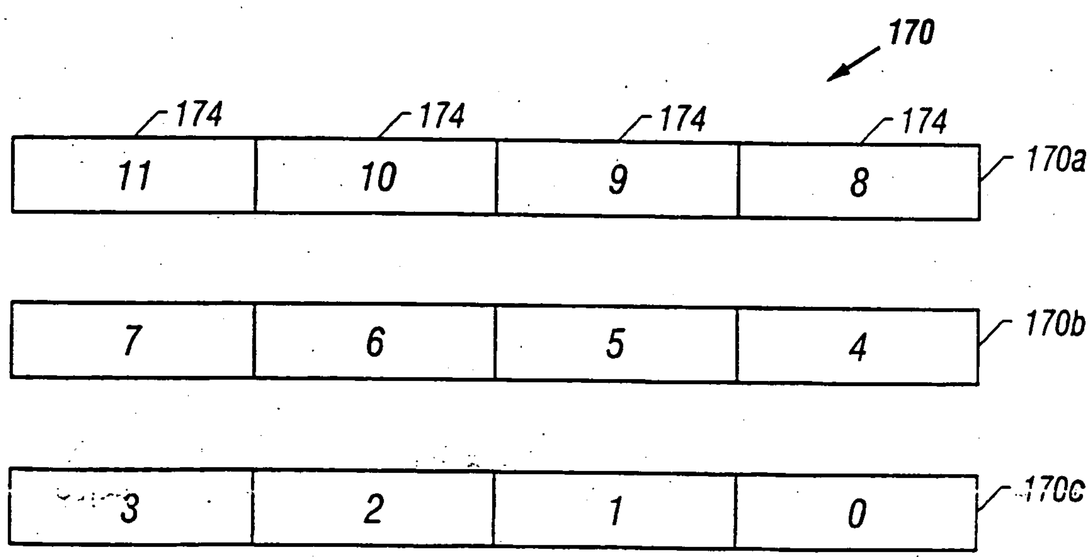

170
174
174
174
174
11
10
9
8
170a
7
6
5
4
170b
3
2
1
0
170c

FIG. 6A

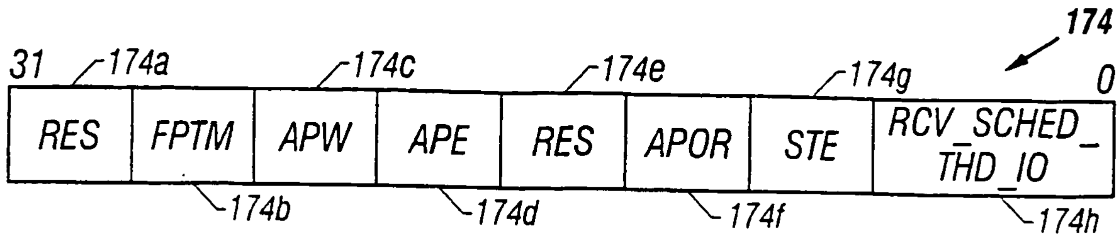


FIG. 6B

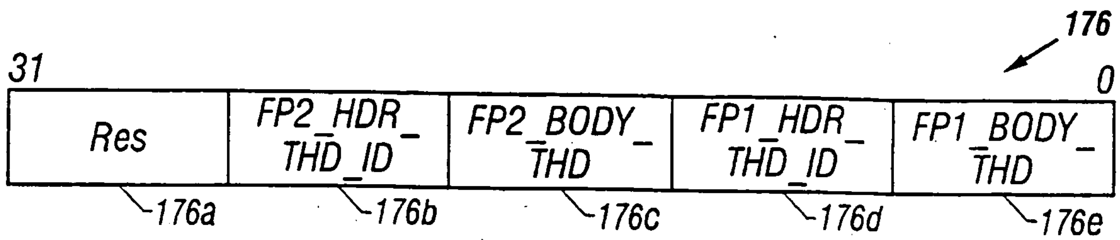


FIG. 6C

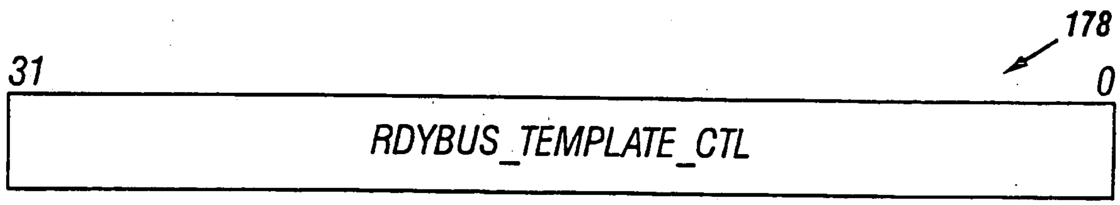


FIG. 6D

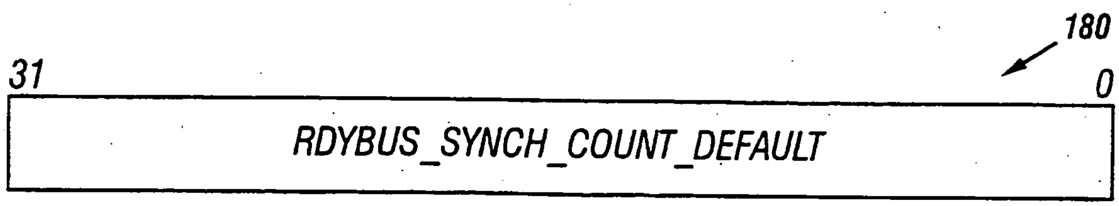


FIG. 6E

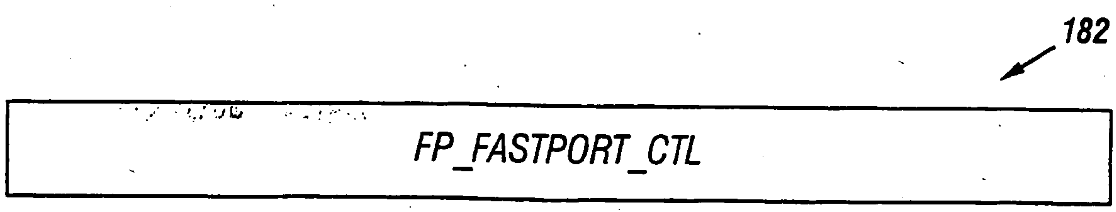


FIG. 6F

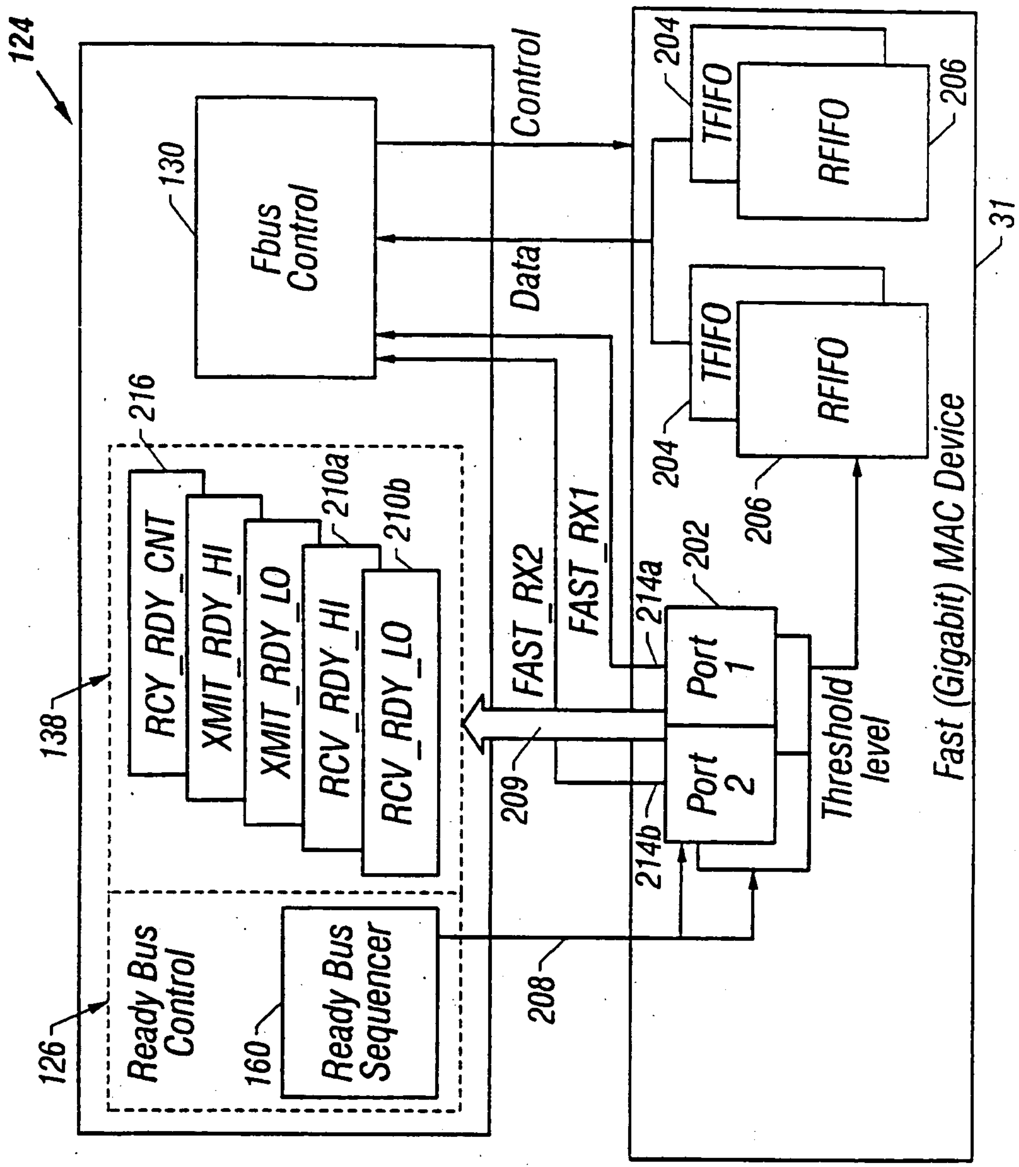

124
130
Fbus Control
Control
Data
TFIFO
RFIFO
204
206
31
216
138
RCY_RDY_CNT
XMIT_RDY_HI
XMIT_RDY_LO
RCV_RDY_HI
RCV_RDY_LO
210a
210b
FAST_RX2
FAST_RX1
214a
202
Port 1
Port 2
209
214b
Threshold level
Fast (Gigabit) MAC Device
126
Ready Bus Control
160
Ready Bus Sequencer
208

FIG. 7

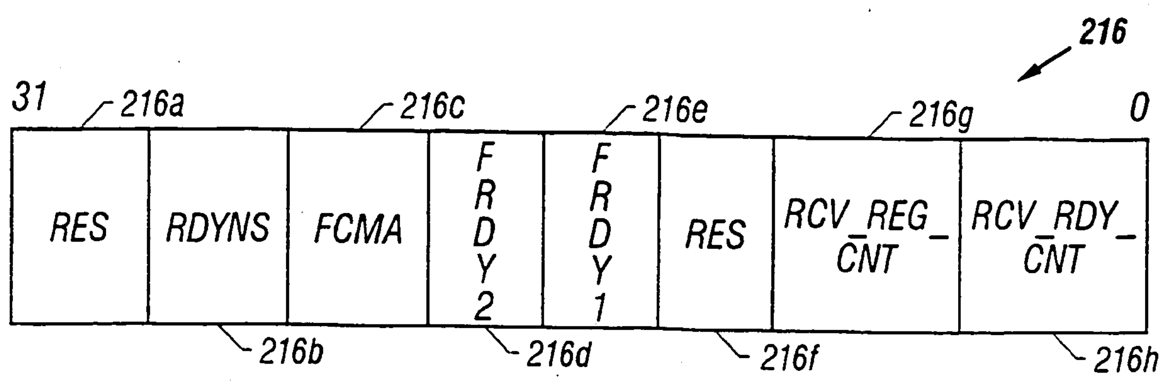


FIG. 8A

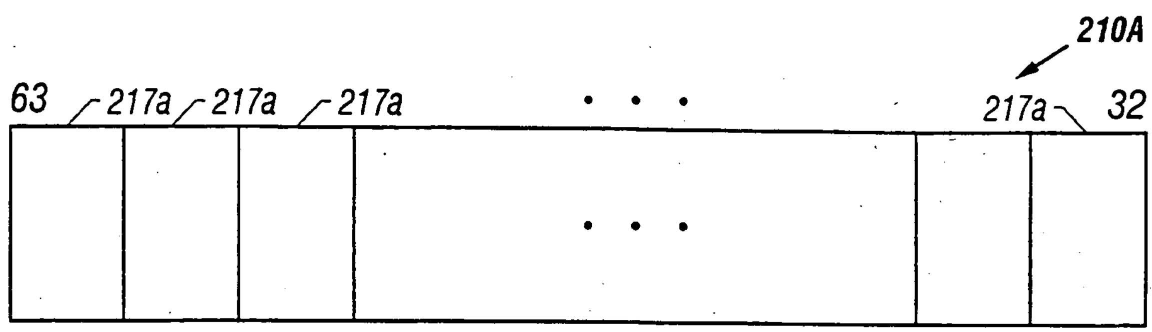


FIG. 8B

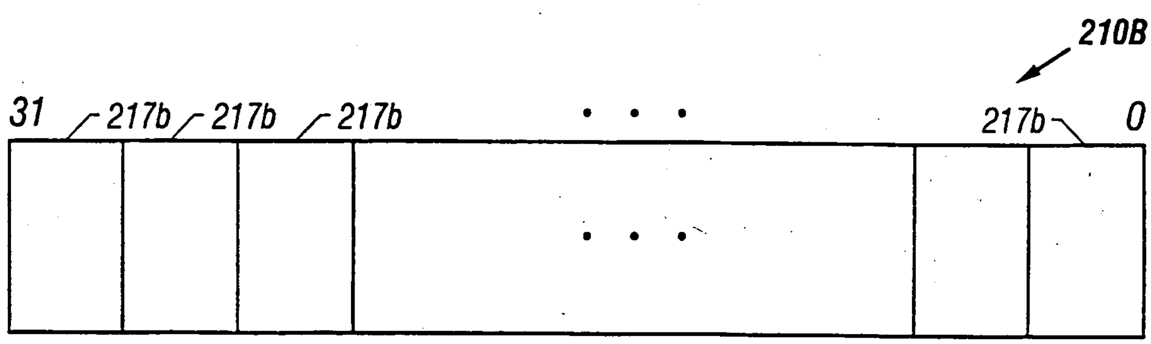
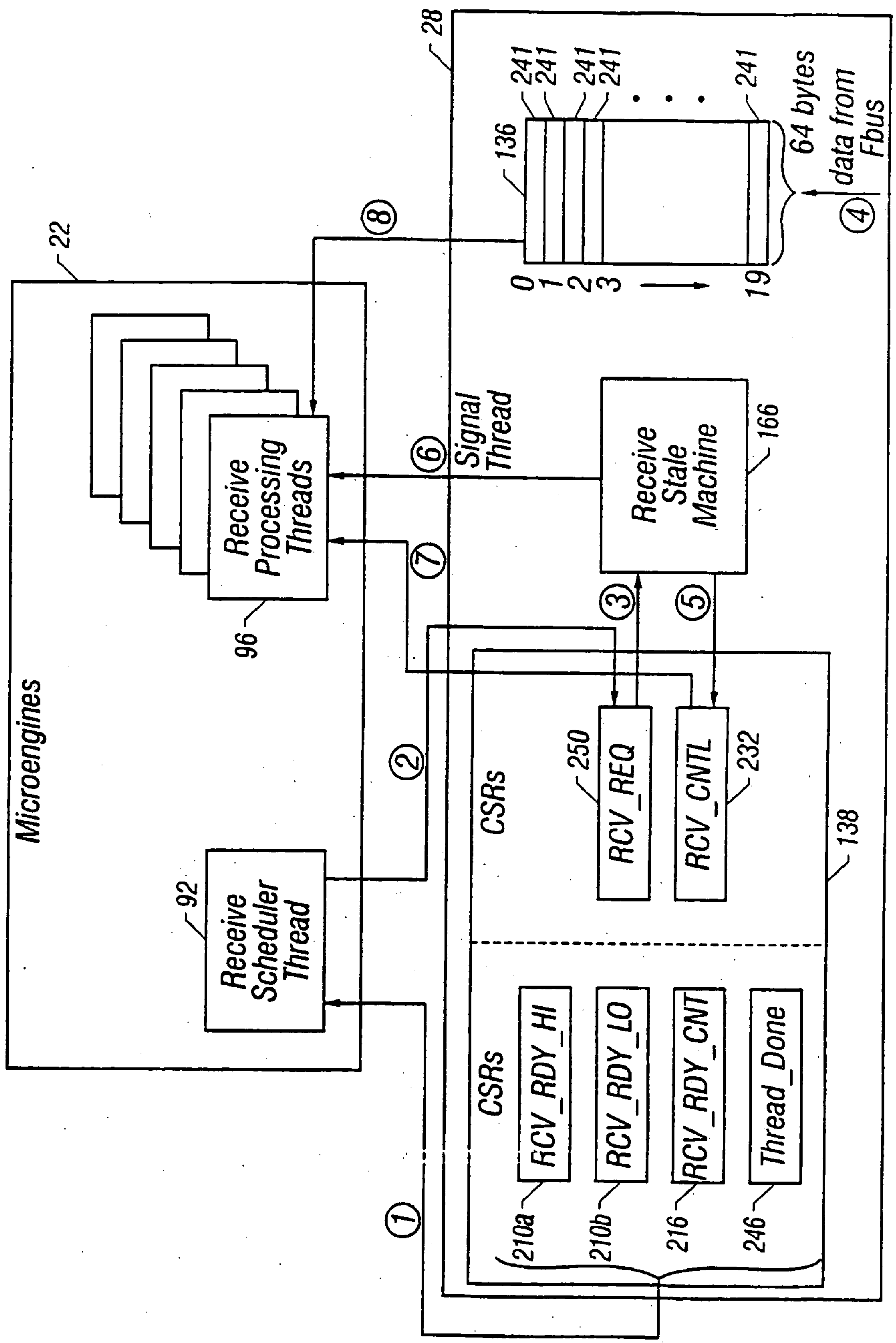


FIG. 8C



22
Microengines
Receive Processing Threads
96
Receive Scheduler Thread
92
28
136
241
0
1
2
3
19
64 bytes data from Fbus
Signal Thread
Receive Stale Machine
166
CSRs
RCV_REQ
250
RCV_CNTL
232
138
CSRs
RCV_RDY_HI
RCV_RDY_LO
RCV_RDY_CNT
Thread_Done
210a
210b
216
246
1
2
3
4
5
6
7
8

FIG. 9

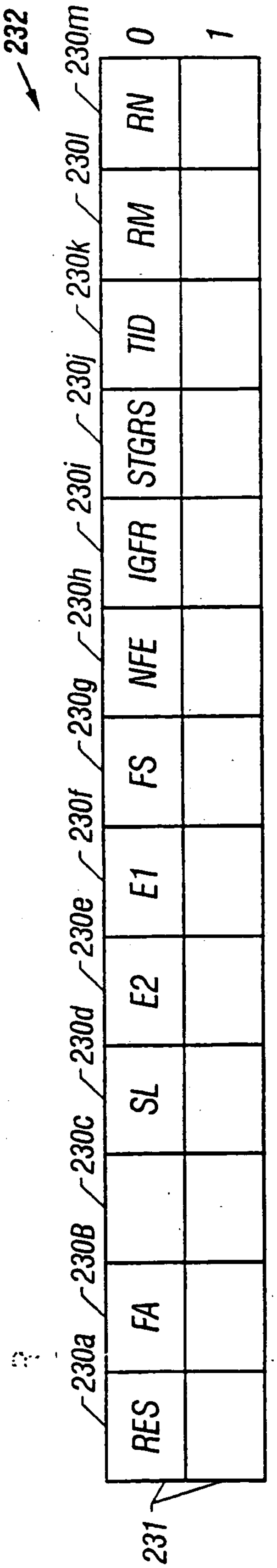


FIG. 10A

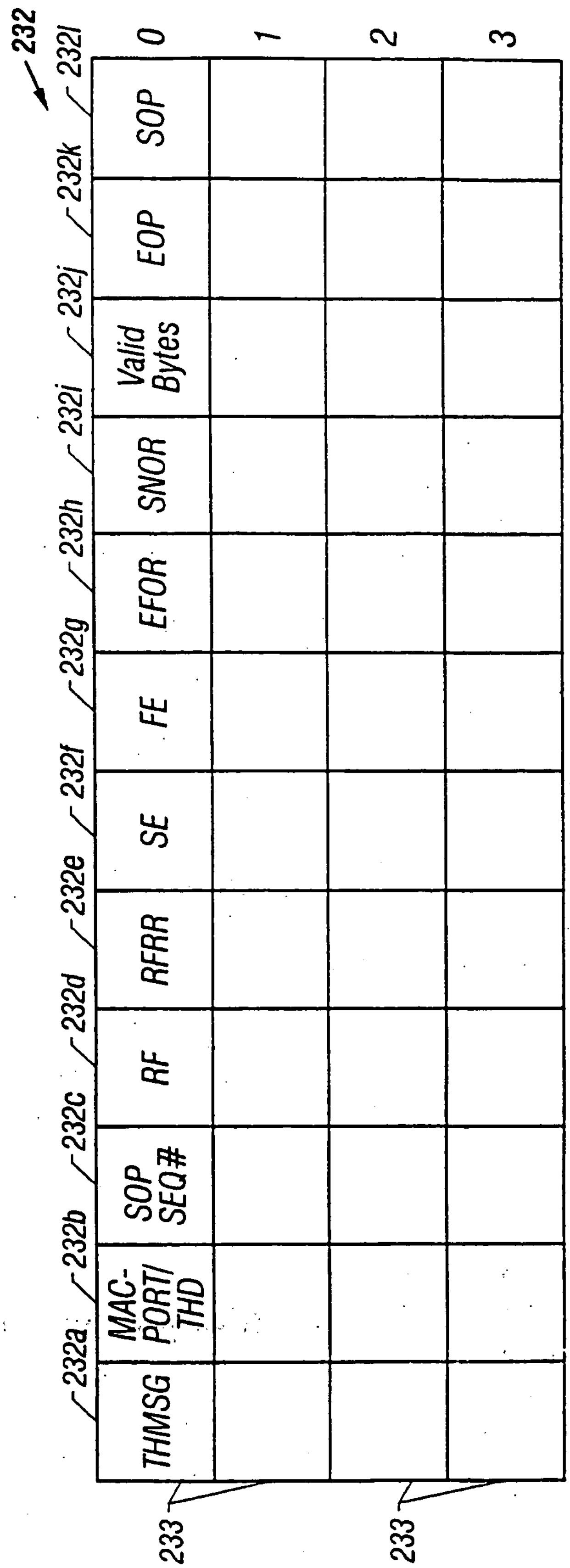


FIG. 10B

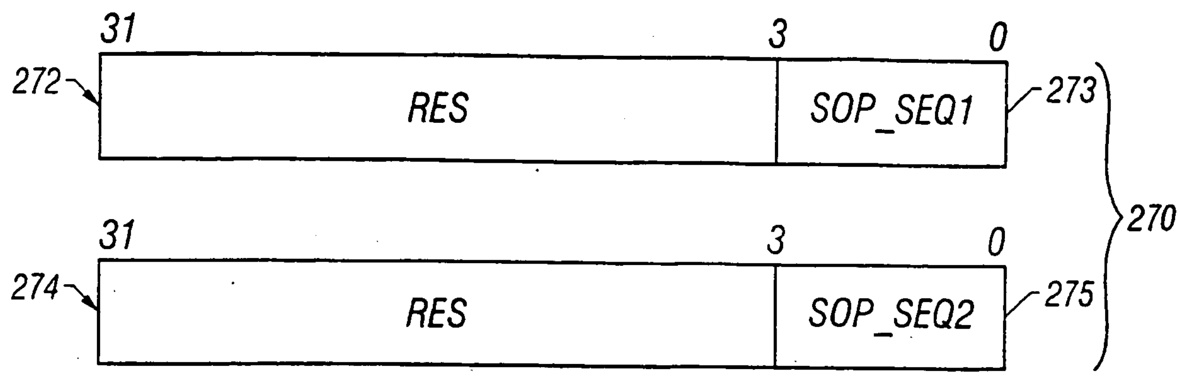


FIG. 11A

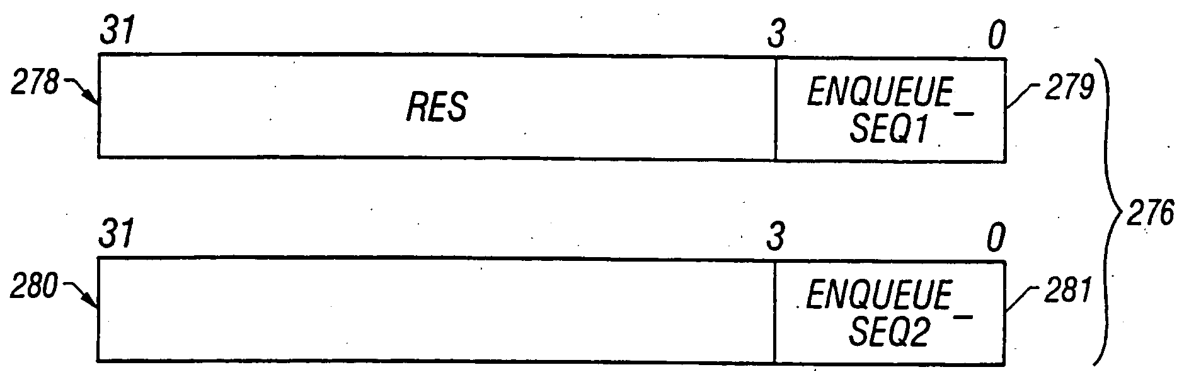


FIG. 11B

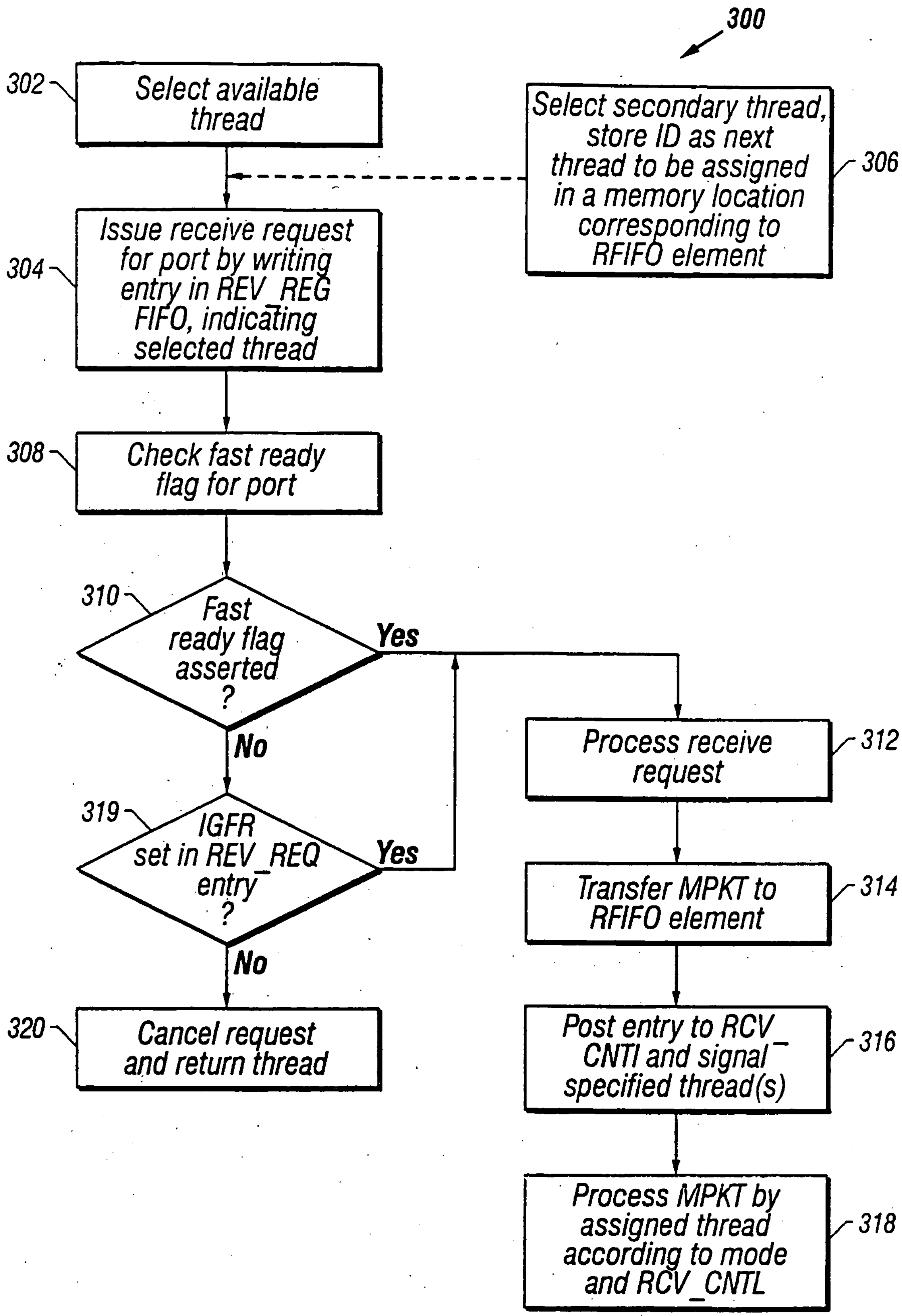

300
302
Select available thread
304
Issue receive request for port by writing entry in REV_REG FIFO, indicating selected thread
306
Select secondary thread, store ID as next thread to be assigned in a memory location corresponding to RFIFO element
308
Check fast ready flag for port
310
Fast ready flag asserted ?
Yes
No
319
IGFR set in REV_REQ entry ?
Yes
No
320
Cancel request and return thread
312
Process receive request
314
Transfer MPKT to RFIFO element
316
Post entry to RCV_CNTl and signal specified thread(s)
318
Process MPKT by assigned thread according to mode and RCV_CNTL

FIG. 12

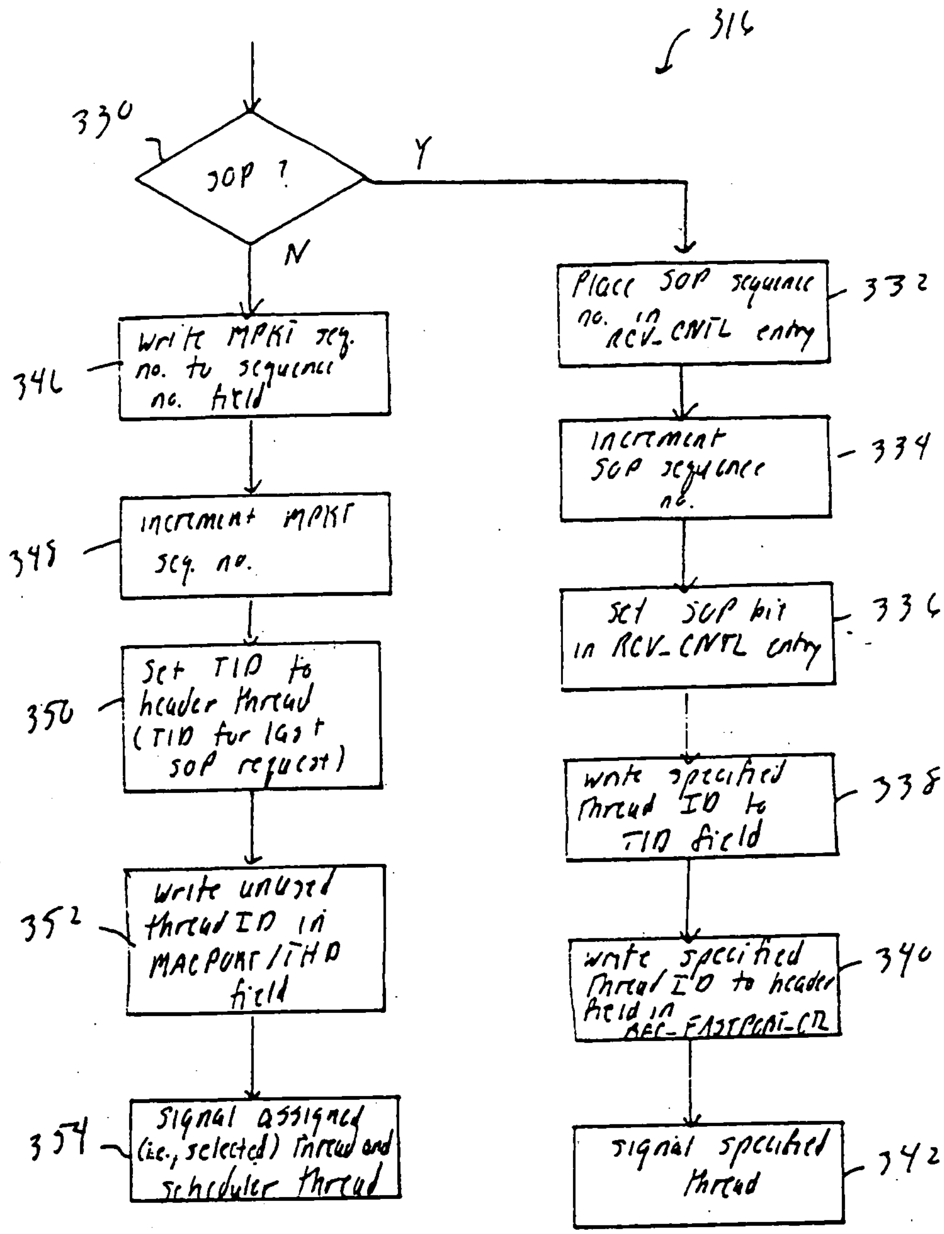

316
330
SOP ?
Y
N
Place SOP sequence no. in RCV_CNTL entry
332
Write MPKT seq. no. to sequence no. field
346
increment SOP sequence no.
334
increment MPKT seq. no.
348
set SOP bit in RCV_CNTL entry
336
Set TID to header thread (TID for last SOP request)
350
write specified thread ID to TID field
338
Write unused thread ID in MACPORT/THD field
352
write specified thread ID to header field in
340
Signal assigned (i.e., selected) thread and scheduler thread
354
signal specified thread
342

FIG. 13A

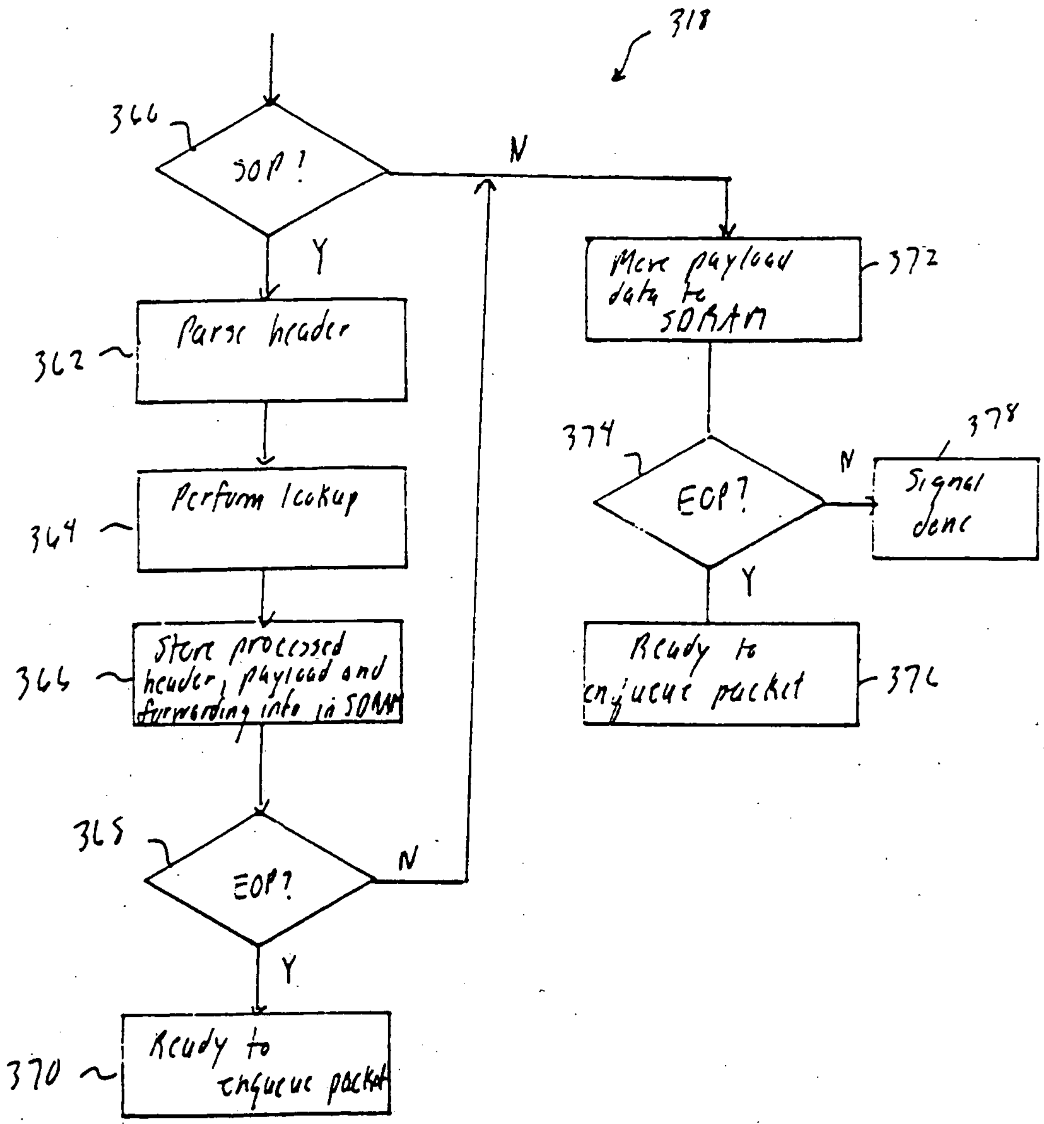

318
360
SOP?
N
Y
Parse header
362
Perform lookup
364
Store processed header, payload and forwarding info in SDRAM
366
368
EOP?
N
Y
Ready to enqueue packet
370
Move payload data to SDRAM
372
374
EOP?
N
Signal done
378
Y
Ready to enqueue packet
376

FIG. 13B

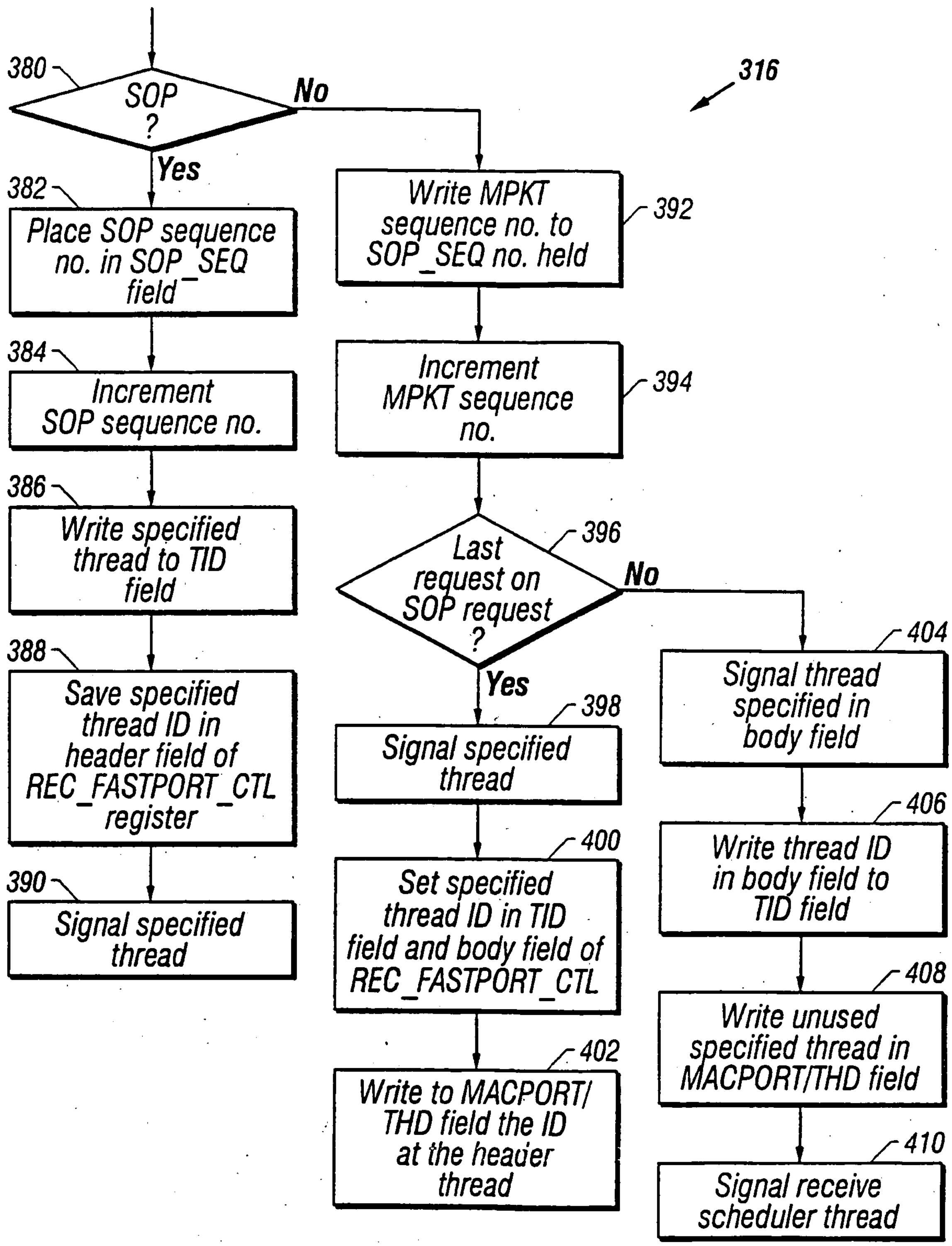

316
380
SOP ?
No
Yes
382
Place SOP sequence no. in SOP_SEQ field
384
Increment SOP sequence no.
386
Write specified thread to TID field
388
Save specified thread ID in header field of REC_FASTPORT_CTL register
390
Signal specified thread
392
Write MPKT sequence no. to SOP_SEQ no. held
394
Increment MPKT sequence no.
396
Last request on SOP request ?
No
Yes
398
Signal specified thread
400
Set specified thread ID in TID field and body field of REC_FASTPORT_CTL
402
Write to MACPORT/THD field the ID at the header thread
404
Signal thread specified in body field
406
Write thread ID in body field to TID field
408
Write unused specified thread in MACPORT/THD field
410
Signal receive scheduler thread

FIG. 14A

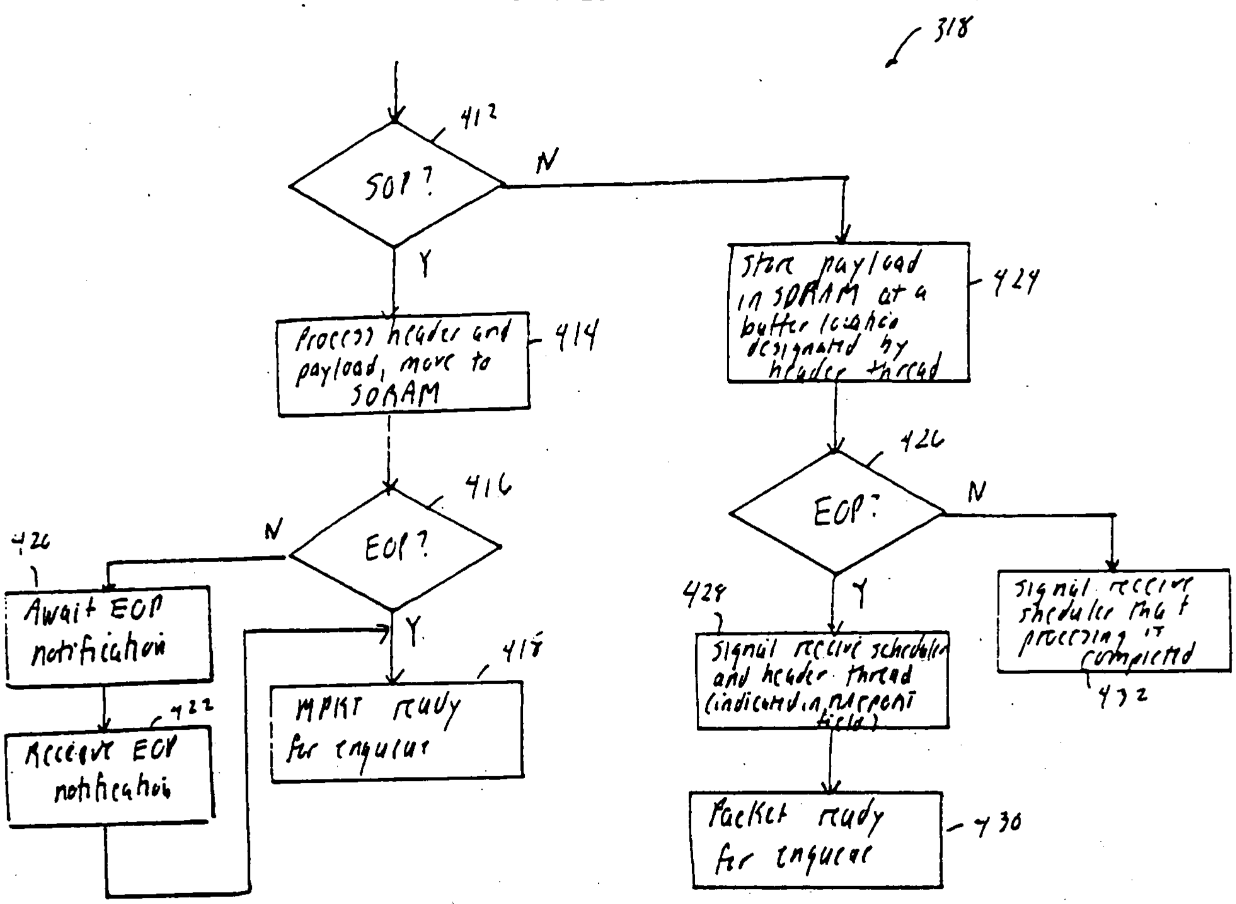

318
412
SOP?
N
Y
Store payload in SDRAM at a buffer location designated by header thread
424
Process header and payload, move to SDRAM
414
426
EOP?
N
416
EOP?
N
420
Await EOP notification
Y
418
MPKT ready for enqueue
422
Receive EOP notification
428
Y
Signal receive scheduler and header thread (indicated in nextpoint field)
Signal receive scheduler that processing is completed
432
Packet ready for enqueue
430

FIG. 14B

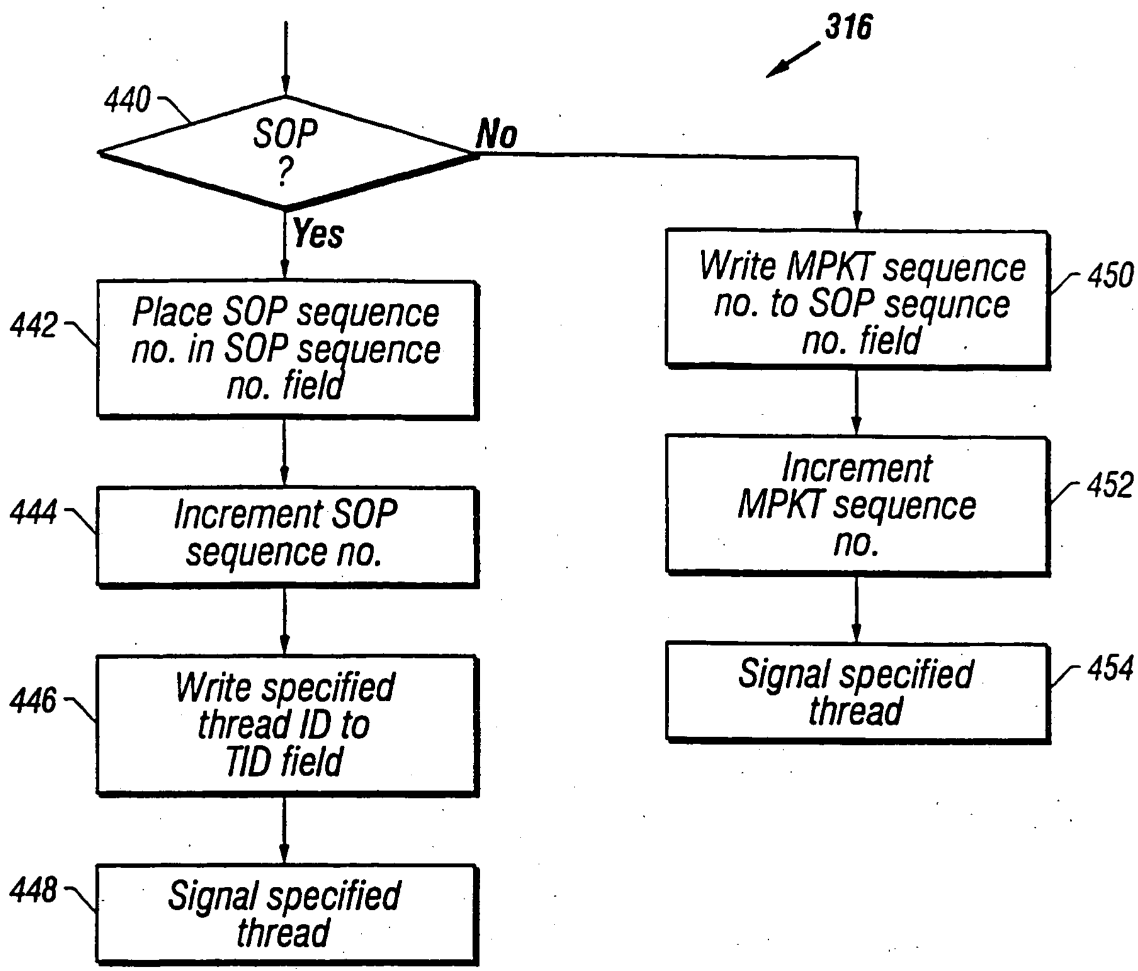

316
440
SOP ?
No
Yes
442
Place SOP sequence no. in SOP sequence no. field
444
Increment SOP sequence no.
446
Write specified thread ID to TID field
448
Signal specified thread
450
Write MPKT sequence no. to SOP sequnce no. field
452
Increment MPKT sequence no.
454
Signal specified thread

FIG. 15A

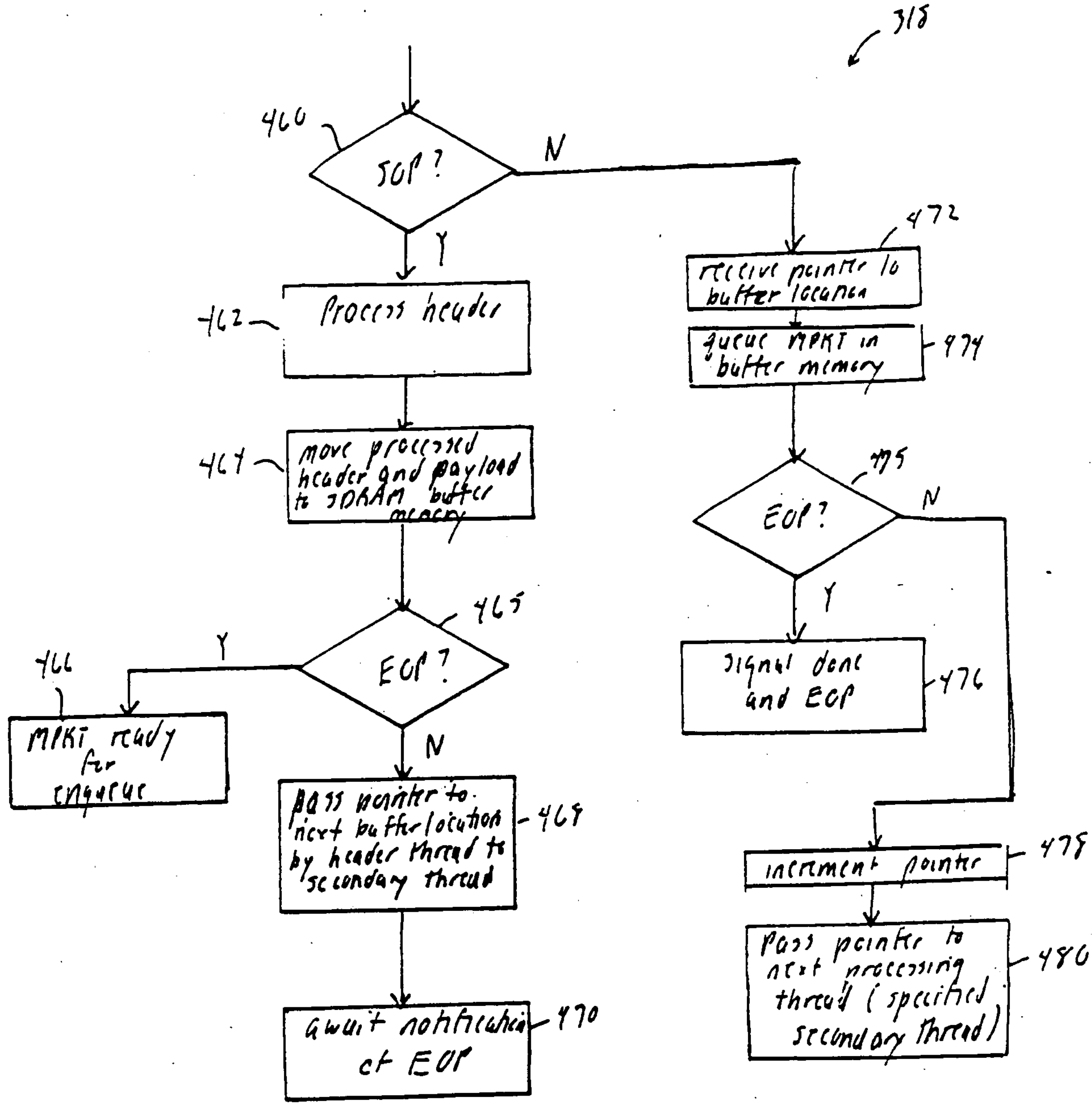

318
460
SOP?
N
Y
462
Process header
464
move processed header and payload to SDRAM buffer memory
465
EOP?
Y
466
MPKT ready for enqueue
N
pass pointer to next buffer location by header thread to secondary thread
468
await notification of EOP
470
472
receive pointer to buffer location
queue MPKT in buffer memory
474
475
EOP?
N
Y
signal done and EOP
476
increment pointer
478
Pass pointer to next processing thread (specified secondary thread)
480

FIG. 15B

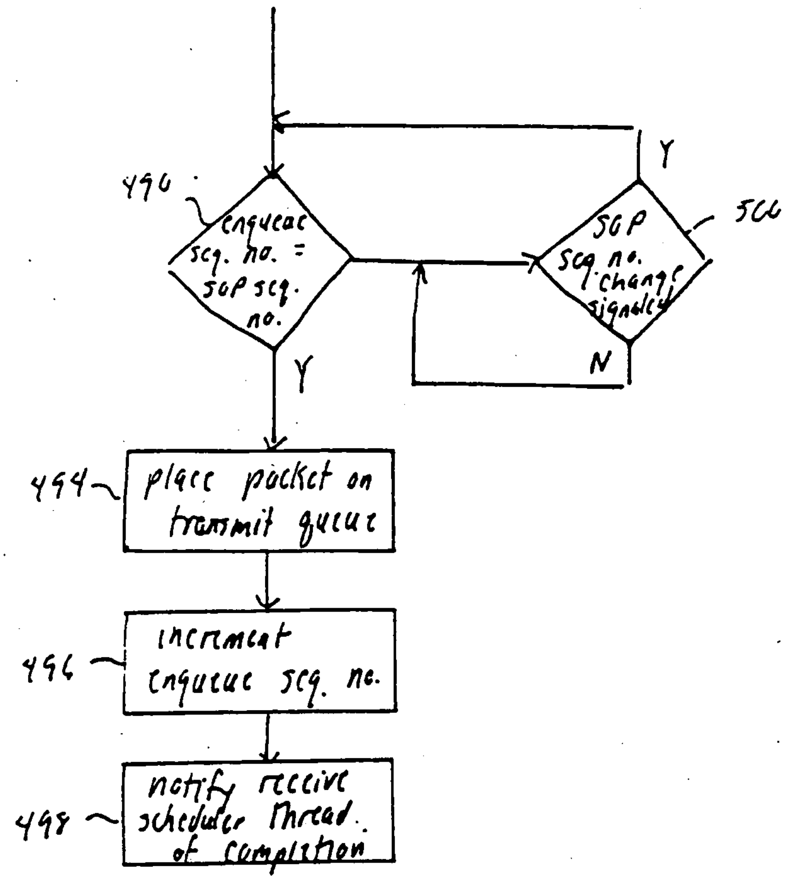

490
enqueue seq. no. = SOP seq. no.
Y
500
SOP seq no. change signaled
Y
N
494
place packet on transmit queue
496
increment enqueue seq. no.
498
notify receive scheduler thread of completion

FIG. 16

(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
12 July 2001 (12.07.2001)

PCT

(10) International Publication Number
WO 01/50679 A3

(51) International Patent Classification[7]: H04L 12/56, 12/44

(21) International Application Number: PCT/US00/33405

(22) International Filing Date: 7 December 2000 (07.12.2000)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
09/474,650 29 December 1999 (29.12.1999) US

(63) Related by continuation (CON) or continuation-in-part (CIP) to earlier application:
US 09/474,650 (CON)
Filed on 29 December 1999 (29.12.1999)

(71) Applicant (for all designated States except US): INTEL CORPORATION [US/US]; 2200 Mission College Boulevard, Santa Clara, CA 95052 (US).

(72) Inventors; and
(75) Inventors/Applicants (for US only): WOLRICH, Gilbert [US/US]; 4 Cider Mill Road, Framingham, MA 01701 (US). BERNSTEIN, Debra [US/US]; 443 Peakham Road, Sudbury, MA 01776 (US). ADILETTA, Matthew, J. [US/US]; 20 Monticello Drive, Worcester, MA 01603 (US). HOOPER, Donald, F. [US/US]; 19 Main Circle, Shrewsbury, MA 01545 (US).

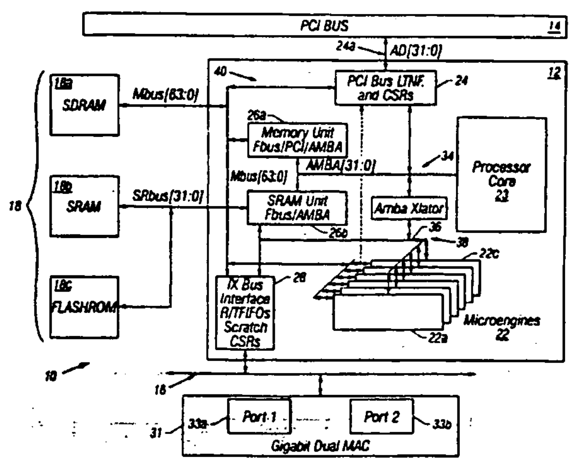
(74) Agent: HARRIS, Scott, C.; Fish & Richardson P.C., 4350 La Jolla Village Drive, Suite 500, San Diego, CA 92122 (US).

(81) Designated States (national): AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CR, CU, CZ, DE, DK, DM, DZ, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, TZ, UA, UG, US, UZ, VN, YU, ZA, ZW.

(84) Designated States (regional): ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE, TR), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GW, ML, MR, NE, SN, TD, TG).

[Continued on next page]

(54) Title: METHOD AND APPARATUS FOR GIGABIT PACKET ASSIGNMENT FOR MULTITHREADED PACKET PROCESSING



(57) Abstract: A network processor that has multiple processing elements, each supporting multiple simultaneous program threads with access to shared resources in an interface. Packet data is received from high-speed ports in segments and each segment is assigned to one of the program threads. Each packet may be assigned to a single program thread, two program threads - one for header segment processing and the other for handling payload segment(s) - or a different program thread for segment of data in a packet. Dedicated inputs for ready status and sequence numbers provide assistance needed for receiving the packet data over a high speed port. The dedicated inputs are used to monitor ready flags from the high speed ports on a cycle-by-cycle basis. The sequence numbers are used by the assigned threads to maintain ordering of segments within a packet, as well as to order the writes of the complete packets to transmit queues.

WO 01/50679 A3

Published:
— *with international search report*

(88) Date of publication of the international search report: 17 January 2002

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

INTERNATIONAL SEARCH REPORT

International Application No
PCT/US 00/33405

A. CLASSIFICATION OF SUBJECT MATTER
IPC 7 H04L12/56 H04L12/44

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC 7 H04L

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)

EPO-Internal, WPI Data, PAJ

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category *	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X Y A	US 5 173 897 A (PFEIFFER BODO ET AL) 22 December 1992 (1992-12-22) abstract column 2, line 49 - line 57 column 3, line 25 - line 31 column 3, line 34 - line 44 column 4, line 17 - line 20 column 4, line 58 - line 61 column 5, line 35 - line 44 claim 1 ---	1-4,25, 26 7-10 5,6, 11-24,27
	-/--	

[X] Further documents are listed in the continuation of box C.

[X] Patent family members are listed in annex.

° Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
16 July 2001	23/07/2001
Name and mailing address of the ISA European Patent Office, P.B. 5818 Patentlaan 2 NL - 2280 HV Rijswijk Tel. (+31-70) 340-2040, Tx. 31 651 epo nl, Fax: (+31-70) 340-3016	Authorized officer Lai, C

6

Form PCT/ISA/210 (second sheet) (July 1992)

INTERNATIONAL SEARCH REPORT

International Application No
PCT/US 00/33405

C.(Continuation) DOCUMENTS CONSIDERED TO BE RELEVANT

Category °	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
Y A	US 5 978 838 A (PARK HEONCHUL ET AL) 2 November 1999 (1999-11-02) abstract column 1, line 14 - line 16 column 2, line 65 -column 3, line 14 column 4, line 24 - line 28 column 8, line 60 - line 66 claim 6 ---	7-10 14,24
X A	EP 0 633 678 A (BELL TELEPHONE MFG) 11 January 1995 (1995-01-11) abstract page 1, column 1 -column 15 page 3, line 15 - line 24 page 6, line 2 - line 11 claim 1 -----	1-4,25, 26 5-13,27

6

INTERNATIONAL SEARCH REPORT

Information on patent family members

International Application No
PCT/US 00/33405

Patent document cited in search report		Publication date	Patent family member(s)			Publication date
US 5173897	A	22-12-1992	DE	3942977	A	27-06-1991
			AT	136706	T	15-04-1996
			AU	639961	B	12-08-1993
			AU	6786890	A	27-06-1991
			CA	2032774	C	01-02-1994
			CA	2032774	A	24-06-1991
			DE	59010278	D	15-05-1996
			EP	0435046	A	03-07-1991
			ES	2088944	T	01-10-1996
			JP	2898762	B	02-06-1999
			JP	6209328	A	26-07-1994
			KR	163180	B	01-12-1998
US 5978838	A	02-11-1999	US	6058465	A	02-05-2000
			JP	10149341	A	02-06-1998
			KR	267091	B	01-11-2000
			CN	1180864	A	06-05-1998
			DE	19735350	A	12-03-1998
			FR	2752630	A	27-02-1998
			JP	10134036	A	22-05-1998
EP 0633678	A	11-01-1995	AU	677840	B	08-05-1997
			AU	6479494	A	12-01-1995
			CA	2126916	A	30-12-1994
			DE	69329059	D	24-08-2000
			DE	69329059	T	22-03-2001
			ES	2148197	T	16-10-2000
			US	5548593	A	20-08-1996

(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
12 July 2001 (12.07.2001)

PCT

(10) International Publication Number
WO 01/50247 A2

(51) International Patent Classification[7]: G06F 9/00

(21) International Application Number: PCT/US00/34537

(22) International Filing Date: 19 December 2000 (19.12.2000)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
09/479,377 5 January 2000 (05.01.2000) US

(63) Related by continuation (CON) or continuation-in-part (CIP) to earlier application:
US 09/479,377 (CON)
Filed on 5 January 2000 (05.01.2000)

(71) Applicant *(for all designated States except US)*: **INTEL CORPORATION** [US/US]; 2200 Mission College Boulevard, Santa Clara, CA 95052 (US).

(72) Inventors; and
(75) Inventors/Applicants *(for US only)*: **WOLRICH, Gilbert** [US/US]; 4 Cider Mill Road, Framingham, MA 01701 (US). **ADILETTA, Matthew, J.** [US/US]; 20 Monticello Drive, Worcester, MA 01603 (US). **WHEELER, William** [US/US]; 9 Darlene Drive, Southborough, MA 01772 (US). **CUTTER, Daniel** [US/US]; 14 Walnut Street, Townsend, MA 01469 (US). **BERNSTEIN, Debra** [US/US]; 443 Peakham Road, Sudbury, MA 01776 (US).

(74) Agent: **HARRIS, Scott, C.**; Fish & Richardson P.C., Suite 500, 4350 La Jolla Village Drive, San Diego, CA 92122 (US).

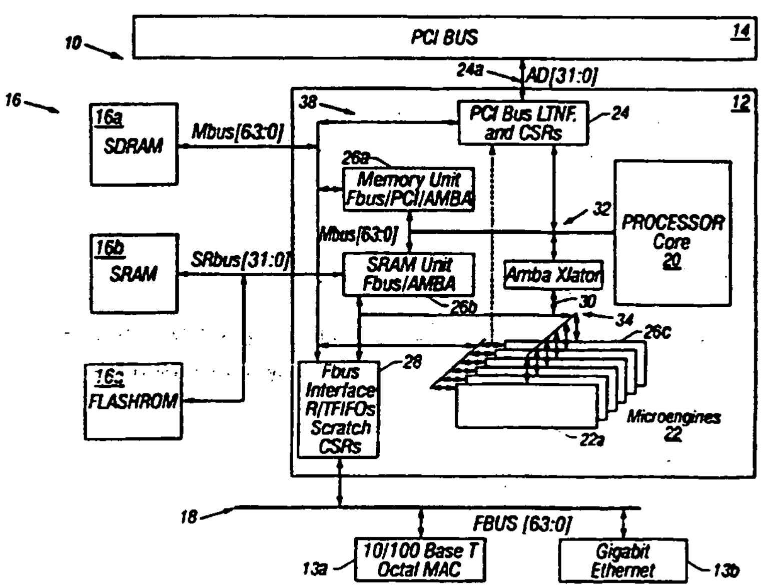
(81) Designated States *(national)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CR, CU, CZ, DE, DK, DM, DZ, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, TZ, UA, UG, US, UZ, VN, YU, ZA, ZW.

(84) Designated States *(regional)*: ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE, TR), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GW, ML, MR, NE, SN, TD, TG).

Published:
— *Without international search report and to be republished upon receipt of that report.*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

(54) Title: MEMORY SHARED BETWEEN PROCESSING THREADS



(57) Abstract: A method includes pushing a datum onto a stack by a first processor and popping the datum off the stack by a second processor.

WO 01/50247 A2

MEMORY SHARED BETWEEN PROCESSING THREADS

BACKGROUND

The invention relates to memory shared between processing threads.

A computer thread is a sequence or stream of computer instructions that performs a task. A computer thread is associated with a set of resources or a context.

SUMMARY

In one general aspect of the invention, a method includes pushing a datum onto a stack by a first processor and popping the datum off the stack by the second processor.

Advantages and other features of the invention will become apparent from the following description and from the claims.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a block diagram of a system employing a hardware-based multi-threaded processor.

FIG. 2 is a block diagram of a MicroEngine employed in the hardware-based multi-threaded processor of FIG. 1.

FIG. 3 is a block diagram showing instruction sets of two threads that are executed on the MicroEngines of FIGS. 1 and 2.

FIG. 4 is a simplified block diagram of the system of FIG. 1 showing selected sub-systems of the processor including a stack module.

FIG. 5A is a block diagram showing the memory components of the stack module of FIG. 4.

FIG. 5B is a block diagram showing the memory components of an alternate implementation of the stack module of FIG. 4.

FIG. 6A is a flow chart of the process of popping a datum from the memory components of FIG. 5A.

FIG. 6B is a block diagram showing the memory components of FIG. 5A after the popping process of FIG. 6A.

FIG. 7A is a flow chart of the process of pushing a datum on the memory components of FIG. 6B.

Fig. 7B is a block diagram showing the memory components of FIG. 6B after the pushing process of FIG. 7A.

FIG. 8 is a block diagram showing memory components used to implement two stacks in one stack module.

DETAILED DESCRIPTION

Referring to FIG. 1, a system 10 includes a parallel, hardware-based multithreaded processor 12. The

hardware-based multithreaded processor 12 is coupled to a bus 14, a memory system 16 and a second bus 18. The bus 14 complies with the Peripheral Component Interconnect Interface, revision 2.1, issued June 1, 1995 (PCI). The system 10 is especially useful for tasks that can be broken into parallel subtasks or functions. Specifically hardware-based multithreaded processor 12 is useful for tasks that are bandwidth oriented rather than latency oriented. The hardware-based multithreaded processor 12 has multiple MicroEngines 22 each with multiple hardware controlled threads that can be simultaneously active and independently work on a task.

The hardware-based multithreaded processor 12 also includes a central controller 20 that assists in loading microcode control for other resources of the hardware-based multithreaded processor 12 and performs other general-purpose computer type functions such as handling protocols, exceptions, and extra support for packet processing where the MicroEngines pass the packets off for more detailed processing such as in boundary conditions. In one embodiment, the processor 20 is a StrongArm (TM) (StrongArm is a trademark of ARM Limited, United Kingdom) based architecture. The general-purpose microprocessor 20 has an operating system. Through the operating system, the processor 20 can call functions to operate on MicroEngines 22a-22f. The processor 20 can use any supported operating system preferably a real time operating system. For the core processor implemented as

a StrongArm architecture, operating systems such as, Microsoft NT real-time, and VXWorks and μC/OS, a freeware operating system available over the Internet at http://www.ucos-ii.com/, can be used.

The hardware-based multithreaded processor 12 also includes a plurality of functional MicroEngines 22a-22f. Functional MicroEngines (MicroEngines) 22a-22f each maintain a plurality of program counters in hardware and states associated with the program counters. Effectively, a corresponding plurality of sets of threads can be simultaneously active on each of the MicroEngines 22a-22f while only one is actually operating at any one time.

In one embodiment, there are six MicroEngines 22a-22f as shown. Each MicroEngines 22a-22f has capabilities for processing four hardware threads. The six MicroEngines 22a-22f operate with shared resources including memory system 16 and bus interfaces 24 and 28. The memory system 16 includes a Synchronous Dynamic Random Access Memory (SDRAM) controller 26a and a Static Random Access Memory (SRAM) controller 26b. SDRAM memory 16a and SDRAM controller 26a are typically used for processing large volumes of data, e.g., processing of network payloads from network packets. The SRAM controller 26b and SRAM memory 16b are used in a networking implementation for low latency, fast access tasks, e.g., accessing look-up tables, memory for the core processor 20, and so forth.

The six MicroEngines 22a-22f access either the SDRAM 16a or SRAM 16b based on characteristics of the data. Thus, low latency, low bandwidth data is stored in and fetched from SRAM, whereas higher bandwidth data for which latency is not as important, is stored in and fetched from SDRAM. The MicroEngines 22a-22f can execute memory reference instructions to either the SDRAM controller 26a or SRAM controller 16b.

Advantages of hardware multithreading can be explained by SRAM or SDRAM memory accesses. As an example, an SRAM access requested by a Thread_0, from a MicroEngine, will cause the SRAM controller 26b to initiate an access to the SRAM memory 16b. The SRAM controller controls arbitration for the SRAM bus, accesses the SRAM 16b, fetches the data from the SRAM 16b, and returns data to a requesting MicroEngine 22a-22b. During an SRAM access, if the MicroEngine e.g., 22a had only a single thread that could operate, that MicroEngine would be dormant until data was returned from the SRAM. By employing hardware context swapping within each of the MicroEngines 22a-22f, the hardware context swapping enables other contexts with unique program counters to execute in that same MicroEngine. Thus, another thread e.g., Thread_1 can function while the first thread, e.g., Thread_0, is awaiting the read data to return. During execution, Thread_1 may access the SDRAM memory 16a. While Thread_1 operates on the SDRAM unit, and Thread_0 is operating on the SRAM unit, a new

thread, e.g., Thread_2 can now operate in the MicroEngine 22a. Thread_2 can operate for a certain amount of time until it needs to access memory or perform some other long latency operation, such as making an access to a bus interface. Therefore, simultaneously, the processor 12 can have a bus operation, SRAM operation and SDRAM operation all being completed or operated upon by one MicroEngine 22a and have one more thread available to process more work in the data path.

The hardware context swapping also synchronizes completion of tasks. For example, two threads could hit the same shared resource e.g., SRAM. Each one of these separate functional units, e.g., the FBUS interface 28, the SRAM controller 26a, and the SDRAM controller 26b, when they complete a requested task from one of the MicroEngine thread contexts reports back a flag signaling completion of an operation. When the MicroEngine receives the flag, the MicroEngine can determine which thread to turn on.

One example of an application for the hardware-based multithreaded processor 12 is as a network processor. As a network processor, the hardware-based multithreaded processor 12 interfaces to network devices such as a media access controller device e.g., a 10/100BaseT Octal MAC 13a or a Gigabit Ethernet device 13b. The Gigabit Ethernet device 13b complies with the IEEE 802.3z standard, approved in June 1998. In general, as a network processor, the hardware-based multithreaded

processor 12 can interface to any type of communication device or interface that receives/sends large amounts of data. Communication system 10 functioning in a networking application could receive a plurality of network packets from the devices 13a, 13b and process those packets in a parallel manner. With the hardware-based multithreaded processor 12, each network packet can be independently processed.

Another example for use of processor 12 is a print engine for a postscript processor or as a processor for a storage subsystem, i.e., RAID disk storage. A further use is as a matching engine. In the securities industry for example, the advent of electronic trading requires the use of electronic matching engines to match orders between buyers and sellers. These and other parallel types of tasks can be accomplished on the system 10.

The processor 12 includes a bus interface 28 that couples the processor to the second bus 18. Bus interface 28 in one embodiment couples the processor 12 to the so-called FBUS 18 (FIFO bus). The FBUS interface 28 is responsible for controlling and interfacing the processor 12 to the FBUS 18. The FBUS 18 is a 64-bit wide FIFO bus, used to interface to Media Access Controller (MAC) devices.

The processor 12 includes a second interface e.g., a PCI bus interface 24 that couples other system components that reside on the PCI 14 bus to the processor

12. The PCI bus interface 24, provides a high-speed data path 24a to memory 16 e.g., the SDRAM memory 16a. Through that path data can be moved quickly from the SDRAM 16a through the PCI bus 14, via direct memory access (DMA) transfers. The hardware based multithreaded processor 12 supports image transfers. The hardware based multithreaded processor 12 can employ a plurality of DMA channels so if one target of a DMA transfer is busy, another one of the DMA channels can take over the PCI bus to deliver information to another target to maintain high processor 12 efficiency. Additionally, the PCI bus interface 24 supports target and master operations. Target operations are operations where slave devices on bus 14 access SDRAMs through reads and writes that are serviced as a slave to target operation. In master operations, the processor core 20 sends data directly to or receives data directly from the PCI interface 24.

Each of the functional units is coupled to one or more internal buses. As described below, the internal buses are dual, 32 bit buses (i.e., one bus for read and one for write). The hardware-based multithreaded processor 12 also is constructed such that the sum of the bandwidths of the internal buses in the processor 12 exceeds the bandwidth of external buses coupled to the processor 12. The processor 12 includes an internal core processor bus 32, e.g., an ASB bus (Advanced System Bus) that couples the processor core 20 to the memory

controller 26a, 26c and to an ASB translator 30 described below. The ASB bus is a subset of the so-called AMBA bus that is used with the Strong Arm processor core. The processor 12 also includes a private bus 34 that couples the MicroEngine units to SRAM controller 26b, ASB translator 30 and FBUS interface 28. A memory bus 38 couples the memory controller 26a, 26b to the bus interfaces 24 and 28 and memory system 16 including flashrom 16c used for boot operations and so forth.

Referring to FIG. 2, an exemplary one of the MicroEngines 22a-22f, e.g., MicroEngine 22f is shown. The MicroEngine includes a control store 70, which, in one implementation, includes a RAM of here 1,024 words of 32 bit. The RAM stores a microprogram. The microprogram is loadable by the core processor 20. The MicroEngine 22f also includes controller logic 72. The controller logic includes an instruction decoder 73 and program counter (PC) units 72a-72d. The four micro program counters 72a-72d are maintained in hardware. The MicroEngine 22f also includes context event switching logic 74. Context event logic 74 receives messages (e.g., SEQ_#_EVENT_RESPONSE; FBI_EVENT_RESPONSE; SRAM _EVENT_RESPONSE; SDRAM _EVENT_RESPONSE; and ASB _EVENT_RESPONSE) from each one of the shared resources, e.g., SRAM 26a, SDRAM 26b, or processor core 20, control and status registers, and so forth. These messages provide information on whether a requested function has completed. Based on whether or not a function requested

by a thread has completed and signaled completion, the thread needs to wait for that completion signal, and if the thread is enabled to operate, then the thread is placed on an available thread list (not shown). The MicroEngine 22f can have a maximum of e.g., 4 threads available.

In addition to event signals that are local to an executing thread, the MicroEngines 22 employ signaling states that are global. With signaling states, an executing thread can broadcast a signal state to all MicroEngines 22. Receive Request Available signal, Any and all threads in the MicroEngines can branch on these signaling states. These signaling states can be used to determine availability of a resource or whether a resource is due for servicing.

The context event logic 74 has arbitration for the four (4) threads. In one embodiment, the arbitration is a round robin mechanism. Other techniques could be used including priority queuing or weighted fair queuing. The MicroEngine 22f also includes an execution box (EBOX) data path 76 that includes an arithmetic logic unit 76a and general-purpose register set 76b. The arithmetic logic unit 76a performs arithmetic and logical functions as well as shift functions. The registers set 76b has a relatively large number of general-purpose registers. As will be described in FIG. 6, in this implementation there are 64 general-purpose registers in a first bank, Bank A and 64 in a second bank, Bank B. The general-purpose

registers are windowed as will be described so that they are relatively and absolutely addressable.

The MicroEngine 22f also includes a write transfer register 78 and a read transfer 80. These registers are also windowed so that they are relatively and absolutely addressable. Write transfer register 78 is where write data to a resource is located. Similarly, read register 80 is for return data from a shared resource. Subsequent to or concurrent with data arrival, an event signal from the respective shared resource e.g., the SRAM controller 26a, SDRAM controller 26b or core processor 20 will be provided to context event arbiter 74 which will then alert the thread that the data is available or has been sent. Both transfer register banks 78 and 80 are connected to the execution box (EBOX) 76 through a data path. In one implementation, the read transfer register has 64 registers and the write transfer register has 64 registers.

Referring to FIG. 3, processor 12 has processing threads 41 and 42 executing in MicroEngines 22a and 22b respectively. In other instances, the threads 41 and 42 may be executed on the same MicroEngine. The processing threads may or may not share data between them. For example, in Fig. 3, processing thread 41 receives data 43 and processes it to produce data 44. Processing thread 42 receives and possesses the data 44 to produce output data 45. Threads 41 and 42 are concurrently active.

Because the MicroEngines 22a and 22b share SDRAM 16a and SRAM 16b (memory), one MicroEngines 22a may need to designate sections of memory for its exclusive use. To facilitate efficient allocation of memory sections, the SDRAM memory is divided into memory segments, referred to as buffers. The memory locations in a buffer share a common address prefix, or pointer. The pointer is used by the processor as an identifier for a buffer.

Pointers to buffers that are not currently in use by a processing thread are managed by pushing the pointers onto a free memory stack. A thread can allocate a buffer for use by the thread by popping a pointer off the stack, and using the pointer to access the corresponding buffer. When a processing thread no longer needs a buffer that is allocated to the processing thread, the thread pushes the pointer to the buffer onto the stack to make the buffer available to other threads.

The threads 41 and 42 have processor instruction sets 46, 47 that respectively include a ''PUSH'' 46a and a ''POP'' 47A instruction. Upon executing either the ''PUSH'' or the ''POP'' instruction, the instruction is transmitted to a logical stack module 56 (FIG. 4).

Referring to Fig. 4, a section of the processor 9 and SRAM 16b provide the logical stack module 56. The logical stack module is implemented as a linked list of SRAM addresses. Each SRAM address on the linked list contains the address of the next item on the list. As a result, if you have the address of the first item on the

list, you can read the contents of that address to find the address of the next item on the list, and so on. Additionally, each address on the linked list is associated with a corresponding memory buffer. Thus the stack module 56 is used to implement a linked list of memory buffers. While in use, the linked list allows the stack to increase or decrease in size as needed.

The stack module 56 includes control logic 51 on the SRAM unit 26b. The control logic 51 performs the necessary operations on the stack while SRAM 16b stores the contents of the stack. One of SRAM registers 50 is used to store the address of the first SRAM location on the stack. The address is also a pointer to the first buffer on the stack.

Although the different components of the stack module 56 and the threads will be explained using an example that uses hardware threads and stack modules, the stack can also be implemented in operating system software threads using software modules. Thread 41 and thread 42 may be implemented as two operating system threads which execute ''PUSH'' and ''POP'' operating system commands to allocate memory from a shared memory pool. The operating system commands may include calls to a library of functions written in the ''C'' programming language. In the operating system example, the equivalents of the control logic 51, the SRAM registers 50 and SRAM 16B are implemented using software within the operating system. The software may be stored in a hard

disk, a floppy disk, computer memory, or other computer readable medium.

Referring to FIG. 5A, SRAM register Q1 stores an address (0xC5) of the first item on the stack 60. The SRAM location (0xC5) of the first item on the stack 60 is used to store the SRAM address (0xA1) of the second item on the stack 60. The SRAM location (0xA1) of the second item on the stack 60 is used to store the address of the third item on the stack 60, etc. The SRAM location (0xE9) of the last item on the stack stores a pre-determined invalid address (0x00), which indicates the end of the linked list.

Additionally, the addresses of the items (0xC5, 0xA1, and 0xE9) on the stack 60 are pointers to stack buffers 61a, 61b, 61c contained within SDRAM 16A. A pointer to a buffer is pushed onto the stack by thread 41, so that the buffer is available for use by other processing threads. A buffer is popped by thread 42 to allocate the buffer for use by thread 42. The pointers are used as an address base to access memory locations in the buffers.

In addition to stack buffers 61a-c, SDRAM 16A also contains processing buffer 62, which is allocated to thread 41. The pointer to processing buffer 62 is not on the stack because it is not available for allocation by other threads. Thread 41 may later push a pointer to the processing buffer 62 onto the stack when it no longer needs the buffer 62.

Although the stack will be discussed with reference to the buffer management scheme above, it can be used without buffers. Referring to Fig. 5B, the SRAM locations 0xC5, 0xA1, and 0xE9 may, respectively, contain data 70a, 70b, and 70c in addition to an address to the next item on the list. Such a scheme may be used to store smaller units of data 70a-c on the stack. In such a scheme, the control logic would assign a memory location within the SRAM for storing the unit of data (datum) that is to be pushed onto the stack. The datum pushed onto the stack may be text, numerical data, or even an address or pointer to another memory location.

Referring to FIG. 6A, to pop a datum off the stack stored in SRAM register Q1, thread 42 executes 101 the instruction ''POP #1''. The pop instruction is part of the instruction set of the MicroEngines 22. The pop instruction is transmitted to control logic 51 over bus 55 for stack processing. Control logic 51 decodes 102 the pop instruction. The control logic also determines 103 the register that contains a pointer to the stack that is referred to in the instruction based on the argument of the pop instruction. Since the argument to the pop instruction is ''#1'', the corresponding register is Q1. The control logic 51 returns 104 the contents of the Q1 register to the context of processing thread 42. The stack of FIG. 5A would return ''0xC5''. Processing thread 42 receives 107 the contents of the Q1 register, which is ''0xC5'', and uses 108 the received content to

access data from the corresponding stack buffer 61b by appending a suffix to the content.

Control logic 27 reads 105 the content (0xA1) of the address (0xC5) stored in the Q1 register. Control logic 27 stores 106 the read content (0xA1) in the Q1 register to indicate that the 0xC5 has been removed from the stack and 0xA1 is now the item at the top of the stack.

Referring to Fig. 6B, the state of the stack after the operations of FIG. 6A will be described. As shown, the register Q1 now contains the address 0xA1, which was previously the address of the second item on the stack. Additionally, the location that was previously stack buffer 61b (in FIG. 5A) is now processing buffer 65, which is used by thread 42. Thus, thread 42 has removed stack buffer 61b from the stack 60 and allocated the buffer 61b for its own use.

Referring to Fig. 7A, the process of adding a buffer to the stack will be described. Thread 41 pushes processing buffer 62 (shown in FIG. 6B) onto the stack by executing 201 the instruction ''PUSH #1 0x01''. The argument 0x01 is a pointer to the buffer 62 because it is a prefix that is common to the address space of the locations in the buffer. The push instruction is transmitted to control logic 51 over the bus 55.

Upon receiving the push instruction, the control logic 51 decodes 202 the instruction and determines 203 the SRAM register corresponding to the instruction, based on the second argument of the push instruction. Since

the second argument is ''#1'', the corresponding register is Q1. The control logic 51 determines the address to be pushed from the third argument (0x01) of the push instruction. The control logic determines 205 the content of the Q1 register by reading the value of the register location. The value 0xA1 is the content of the Q1 register in the stack of FIG. 6B. The control logic stores 206 the content (0xA1) of the Q1 register in the SRAM location whose address is the push address (0x01). The control logic then stores 207 the push address (0x01) in the Q1 register.

Referring to FIG. 7B, the contents of the stack after the operations of FIG. 7A will be described. As shown, the SRAM register Q1, contains the address of the first location on the stack, which is now 0x01. The address of the first location on the stack is also the address of stack buffer 61d, which was previously a processing buffer 62 used by thread 41. The location 0xA1, which was previously the first item on the stack, is now the second item on the stack. Thus, thread 41 adds stack buffer 61d onto the stack to make it available for allocation to other threads. Thread 42 can later allocate the stack buffer 61d for its own use by popping it off the stack, as previously described for FIG. 6A.

Referring to Fig. 8, a second stack 60b (shown in phantom) may be implemented in the same stack module by using a second SRAM control register to store the address of the first element in the second stack 60b. The second

stack may be used to manage a separate set of memory buffers, for example, within SRAM 16b or SDRAM 16a. A first stack 60a has the address of the first element on the stack 60a stored in SRAM register Q1. Additionally, a second stack 60b has the address of its first element stored in register Q6. The first stack 60a is identical to the stack 60 in Fig. 7B. The second stack 60b is similar to previously described stacks.

Other embodiments are within the scope of the following claims. Although the stack 60 (shown in FIG. 5A) stores the pointer to the first element in a register Q1, the linked list in SRAM 16B and the buffers in SDRAM 16A, any of the stack module elements could be stored in any memory location. For example, they could all be stored in SRAM 16b or SDRAM 16a.

Other embodiments my implement the stack in a continuous address space, instead of using a linked list. The size of the buffers may be varied by using pointers (address prefixes) of varying length. For example, a short pointer is a prefix to more addresses and is, therefore, a pointer to a larger address buffer.

Alternatively, the stack may be used to manage resources other than buffers. One possible application of the stack might be to store pointers to the contexts of active threads that are not currently operating. When MicroEngine 22a temporarily sets aside a first active thread to process a second active thread, it stores the context of the first active thread in a memory buffer and

pushes a pointer to that buffer on the stack. Any MicroEngine can resume the processing of the first active thread by popping the pointer to memory buffer containing the context of the first thread and loading that context. Thus the stack can be used to manage the processing of multiple concurrent active threads by multiple processing engines.

What is claimed is:

1. A method comprising:

pushing a datum onto a stack by a first processing thread; and

popping the datum off the stack by a second processing thread.

2. The method of claim 1 wherein the pushing comprises:

executing a push command on the first processing thread, the push command having at least one argument,

determining a pointer to a current stack datum,

determining a location associated with an argument of the push command,

storing the determined pointer at the determined location,

producing a pointer associated with determined location the pointer to the current stack datum.

3. The method of claim 2 wherein determining a location comprises:

decoding the push command.

4. The method of claim 2 wherein determining a location comprises:

storing an argument of the pop command in a location associated with the argument of the push command.

5. The method of claim 2 wherein said push command is at least one of a processor instruction, and an operating system call.

6. The method of claim 1 wherein popping comprises:

executing a pop command by the second processing thread,

determining a pointer to a current stack datum,

returning the determined pointer to the second processing thread,

retrieving a pointer to a previous stack datum from a location associated with the pointer to the current stack datum, and

assigning the retrieved pointer the pointer to the current stack datum.

7. The method of claim 6 wherein the location associated with the pointer to the current stack datum is the location that has an address equal to the value of the pointer to the current stack datum.

8. The method of claim 6 wherein the location associated with the pointer to the current stack datum is the location that has an address equal to the sum of an offset and the value of the pointer to the current stack datum.

9. The method of claim 6 wherein the pop command is at least one of a processor instruction or an operating system call.

10. The method of claim 1 further comprising:

storing data in a memory buffer that is accessible using a buffer pointer having the datum that is pushed onto the stack.

11. The method of claim 1 further comprising:

using the popped datum as a buffer pointer to access information stored in a memory buffer.

12. The method of claim 1 further comprising:

a third processing thread pushing a second datum onto the stack.

13. The method of claim 1 further comprising:

a third processing thread popping a second datum of the stack.

14. A system comprising:

a stack module that stores data by pushing it onto the stack and processing threads can retrieve information by popping the information off the stack,

a first processing thread having a first command set, including at least one command for pushing data onto the stack, and

a second processing thread having a second command

set, including at least one command for popping the data off the stack.

15. The system of claim 14 wherein the first and second processing threads are executed on a single processing engine.

16. The system of claim 14 wherein the first and second processing threads are executed on separate processing engines.

17. The system of claim 16 wherein the separate processing engines are implemented on the same integrated circuit.

18. The system of claim 14 wherein the stack module and the processing threads are on the same integrated circuit.

19. The system of claim 14 where the first and second command sets are at least one of a processor instruction set and an operating system instruction set.

20. The system of claim 14 further comprising a bus interface for communicating between at least one of the processing threads and the stack module.

21. A stack module comprising:

control logic that responds to commands from at least two processing threads, the control logic storing datum on a stack structure in response to a push command

and retrieving datum from the stack in response to a pop command.

22. The stack module of claim 21 further comprising a stack pointer associated with the most recently stored datum on the stack.

23. The stack module of claim 22 further comprising a memory location associated with a first datum on the stack, the second memory location including:

a pointer associated with a second datum which was stored on the stack prior to said first datum.

24. The stack module of claim 22 further comprising a second stack pointer associated with the most recently stored datum on a second stack.

25. The stack module of claim 22 wherein the stack pointer is a register on a processor.

26. The stack module of claim 23 wherein said memory location includes SRAM memory.

27. The stack module of claim 21 wherein the commands are processor instructions.

28. The stack module of claim 21 wherein the commands are operating system instructions.

29. An article comprising a computer-readable medium which stores computer logic, the computer logic comprising:

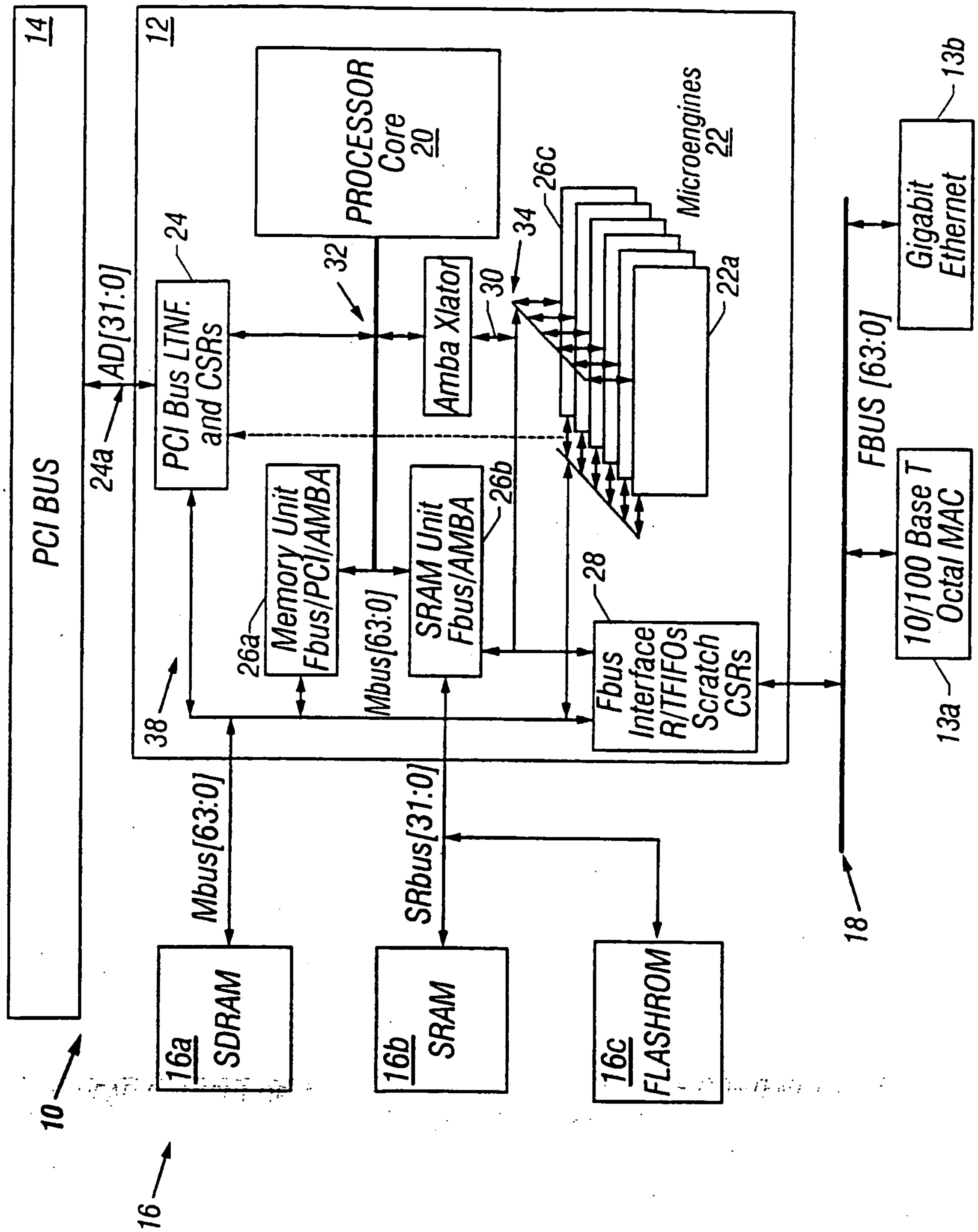
a stack module configured to store data from a first processing thread by pushing the data onto a stack and to retrieve the data for a second processing thread by popping the data off the stack, the stack module being responsive to a first processing thread command to store data on the stack and a second processing thread command to retrieve data from the stack.

30. An article comprising a computer-readable medium which stores computer-executable instructions, the instructions causing a processor to:

store data from a first processing thread by executing an instruction to push the data onto the stack; and

retrieve the data for a second processing thread by executing an instruction to pop the data from the stack for use by the second thread.

FIG. 1A


PCI BUS
14
12
PROCESSOR Core 20
Microengines 22
26c
34
22a
32
30
Amba Xlator
PCI Bus LTNF. and CSRs
24
AD[31:0]
24a
Memory Unit Fbus/PCI/AMBA
26a
SRAM Unit Fbus/AMBA
26b
Mbus[63:0]
28
Fbus Interface R/TFIFOs Scratch CSRs
38
FBUS [63:0]
Gigabit Ethernet
13b
10/100 Base T Octal MAC
13a
Mbus[63:0]
SRbus[31:0]
18
16a SDRAM
16b SRAM
16c FLASHROM
10
16

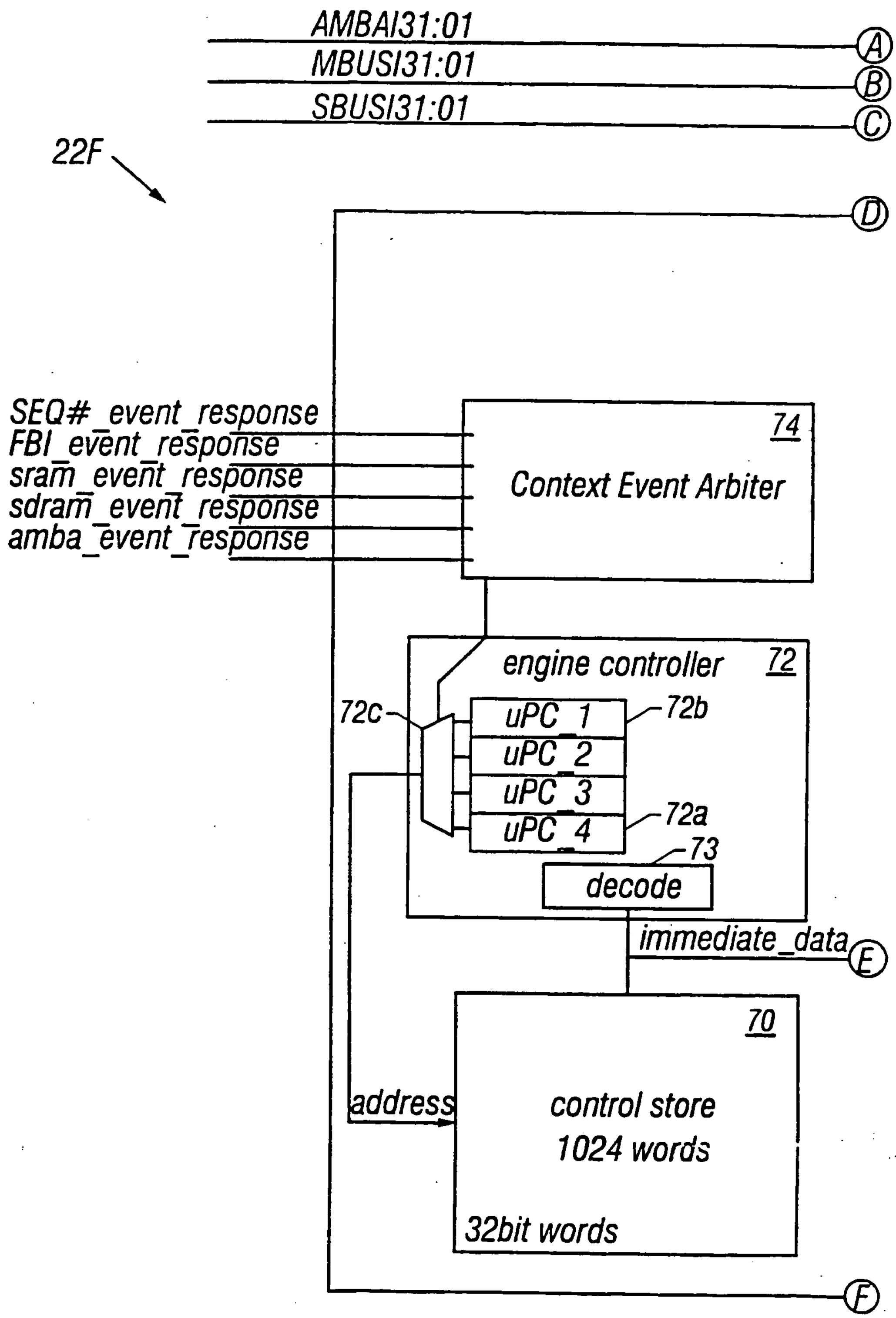

AMBAI31:01
MBUSI31:01
SBUSI31:01
A
B
C
22F
D
SEQ#_event_response
FBI_event_response
sram_event_response
sdram_event_response
amba_event_response
74
Context Event Arbiter
engine controller
72
72c
uPC_1
uPC_2
uPC_3
uPC_4
72b
72a
73
decode
immediate_data
E
70
address
control store
1024 words
32bit words
F

FIG. 2A

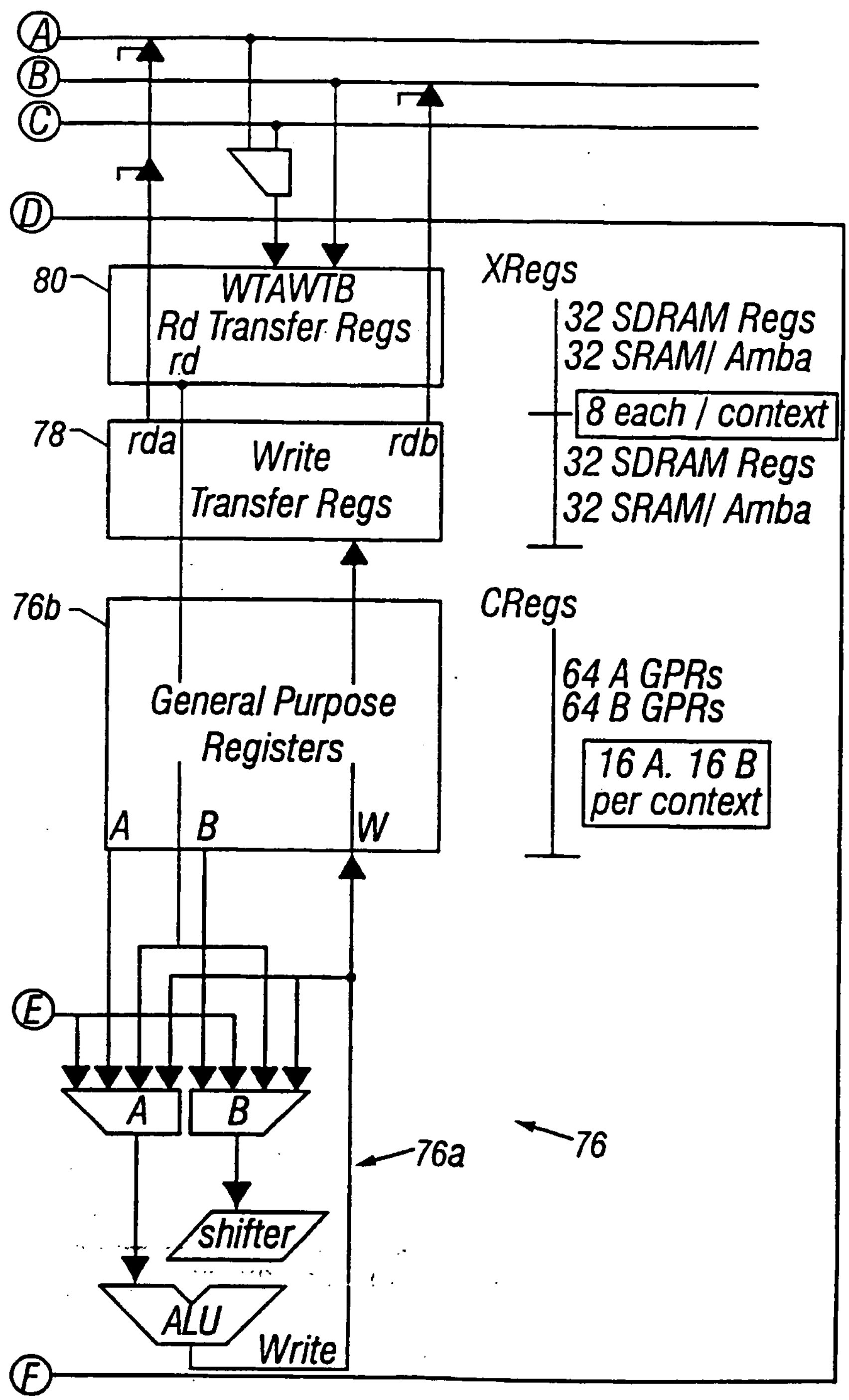

A
B
C
D
E
F
80
WTAWTB
Rd Transfer Regs
rd
78
rda
Write
Transfer Regs
rdb
XRegs
32 SDRAM Regs
32 SRAM/ Amba
8 each / context
32 SDRAM Regs
32 SRAM/ Amba
76b
General Purpose
Registers
A
B
W
CRegs
64 A GPRs
64 B GPRs
16 A. 16 B
per context
A
B
shifter
ALU
Write
76a
76

FIG. 2B

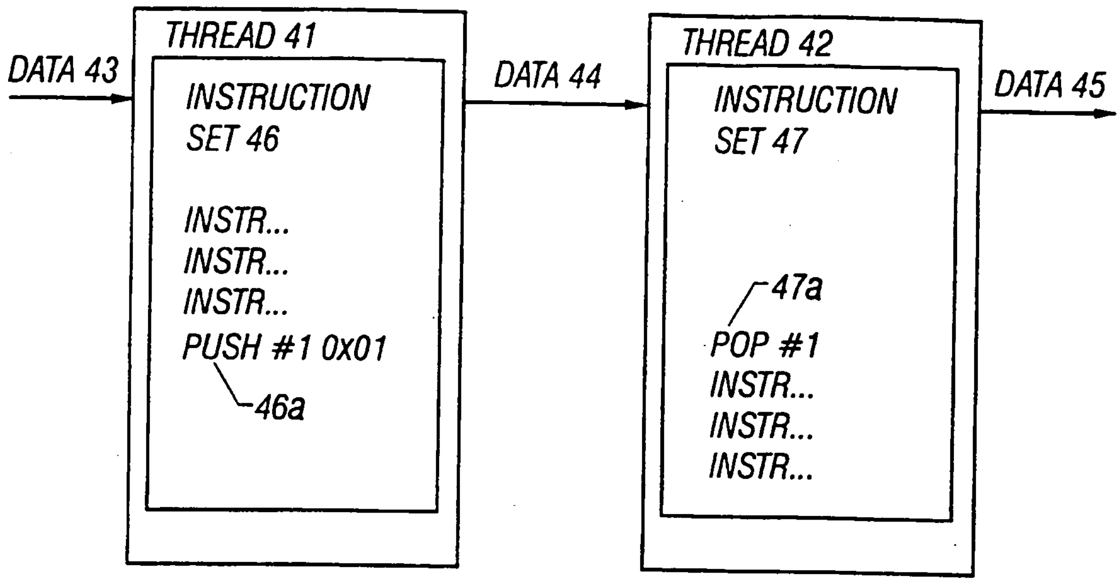

DATA 43
THREAD 41
INSTRUCTION SET 46
INSTR...
INSTR...
INSTR...
PUSH #1 0x01
46a
DATA 44
THREAD 42
INSTRUCTION SET 47
47a
POP #1
INSTR...
INSTR...
INSTR...
DATA 45

FIG. 3

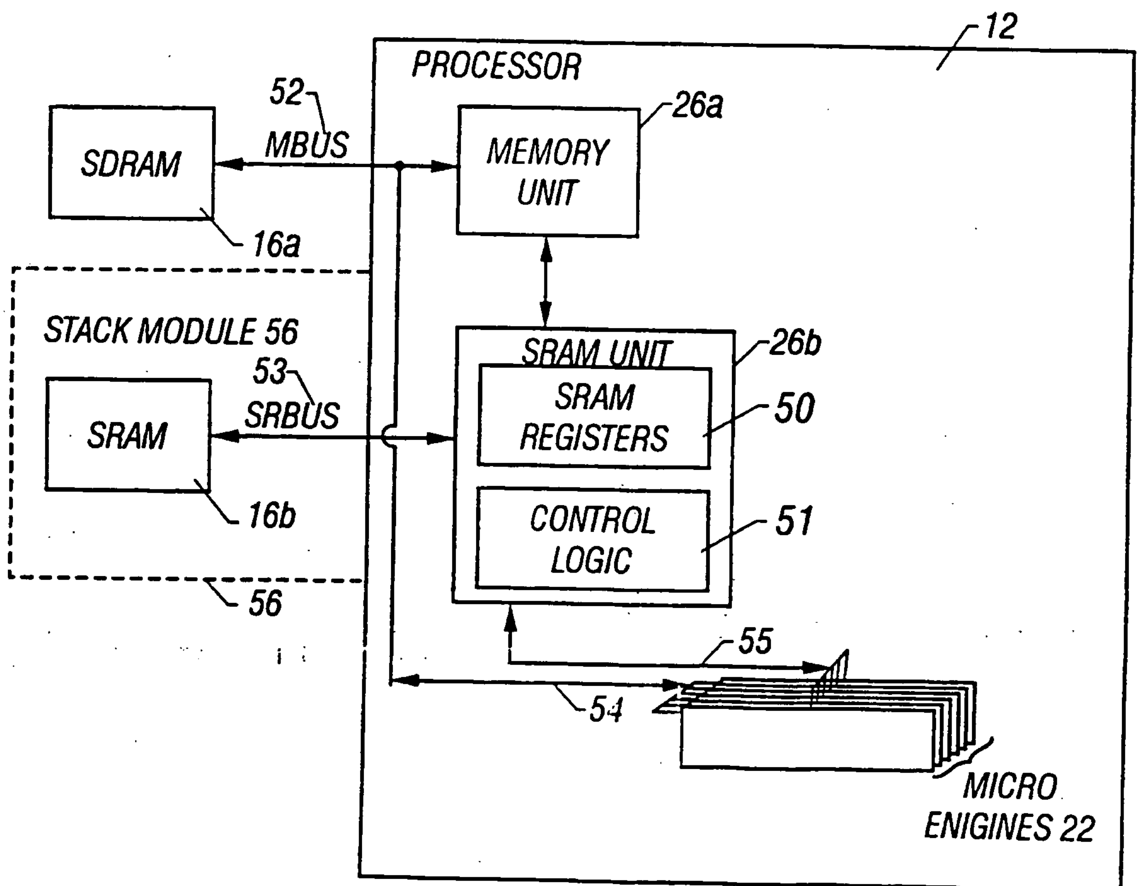
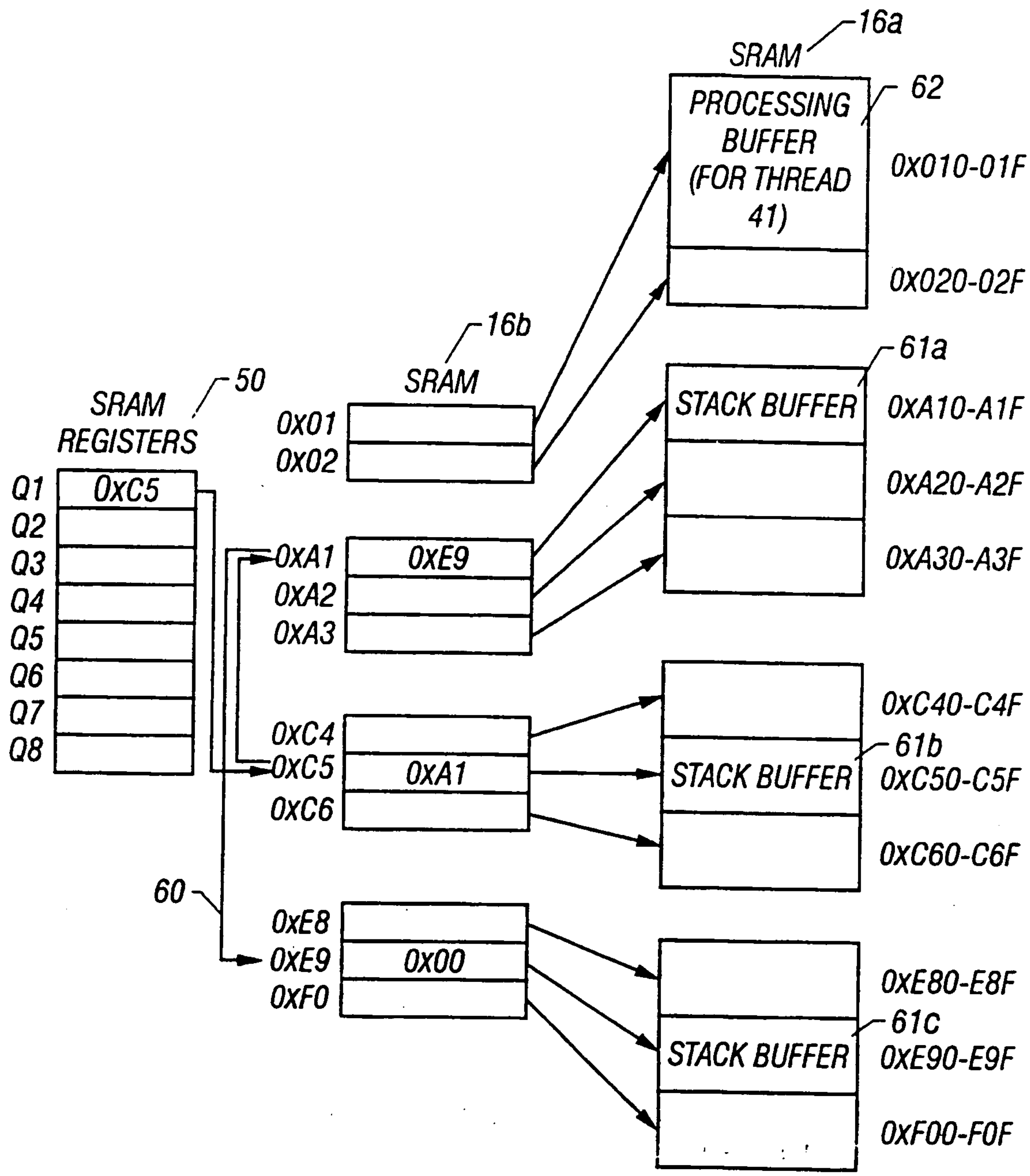

12
PROCESSOR
52
MBUS
SDRAM
16a
MEMORY UNIT
26a
STACK MODULE 56
53
SRBUS
SRAM
16b
56
SRAM UNIT
26b
SRAM REGISTERS
50
CONTROL LOGIC
51
55
54
MICRO ENIGINES 22

FIG. 4



SRAM
16a
PROCESSING BUFFER (FOR THREAD 41)
62
0x010-01F
0x020-02F
16b
SRAM
61a
STACK BUFFER
0XA10-A1F
0XA20-A2F
0XA30-A3F
SRAM REGISTERS
50
0x01
0x02
Q1
0xC5
Q2
Q3
Q4
Q5
Q6
Q7
Q8
0XA1
0XE9
0XA2
0XA3
0XC4
0XC5
0XA1
0XC6
0XC40-C4F
61b
STACK BUFFER
0XC50-C5F
0XC60-C6F
60
0XE8
0XE9
0X00
0XF0
0XE80-E8F
61c
STACK BUFFER
0XE90-E9F
0XF00-F0F

FIG. 5A

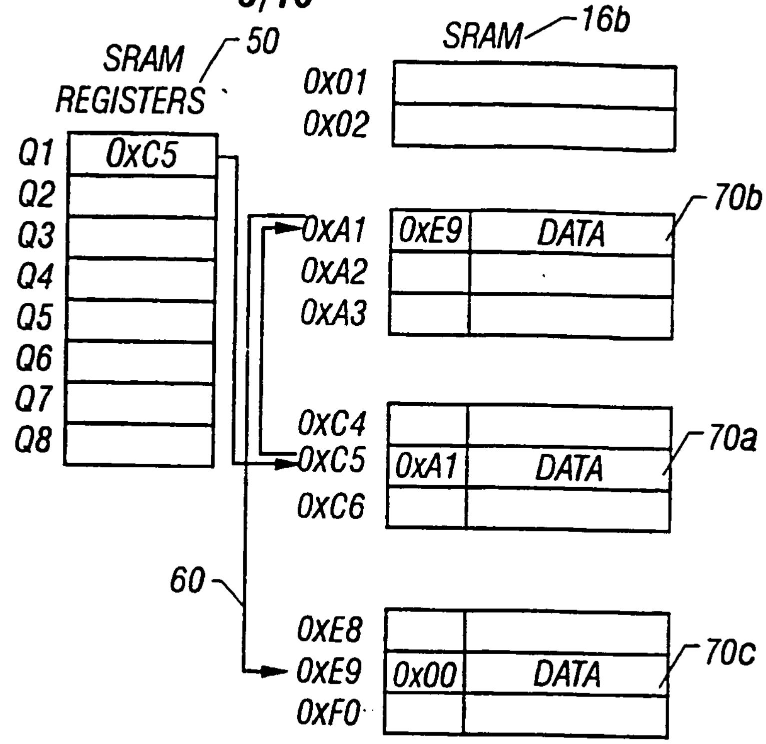

SRAM REGISTERS
50
Q1
Q2
Q3
Q4
Q5
Q6
Q7
Q8
0xC5
60
SRAM
16b
0x01
0x02
0xA1
0xA2
0xA3
0xE9
DATA
70b
0xC4
0xC5
0xC6
0xA1
DATA
70a
0xE8
0xE9
0xF0
0x00
DATA
70c

FIG. 5B

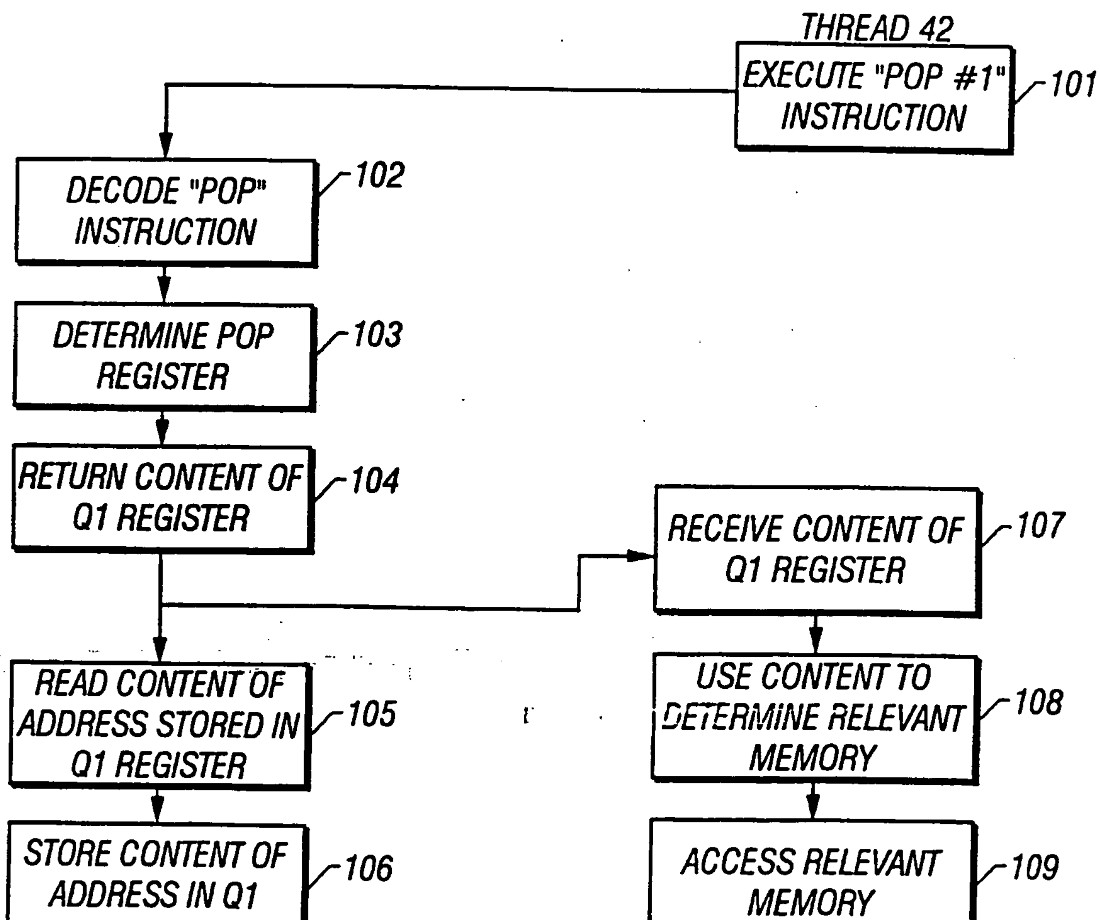

THREAD 42
EXECUTE "POP #1" INSTRUCTION
101
DECODE "POP" INSTRUCTION
102
DETERMINE POP REGISTER
103
RETURN CONTENT OF Q1 REGISTER
104
READ CONTENT OF ADDRESS STORED IN Q1 REGISTER
105
STORE CONTENT OF ADDRESS IN Q1
106
RECEIVE CONTENT OF Q1 REGISTER
107
USE CONTENT TO DETERMINE RELEVANT MEMORY
108
ACCESS RELEVANT MEMORY
109

FIG. 6A

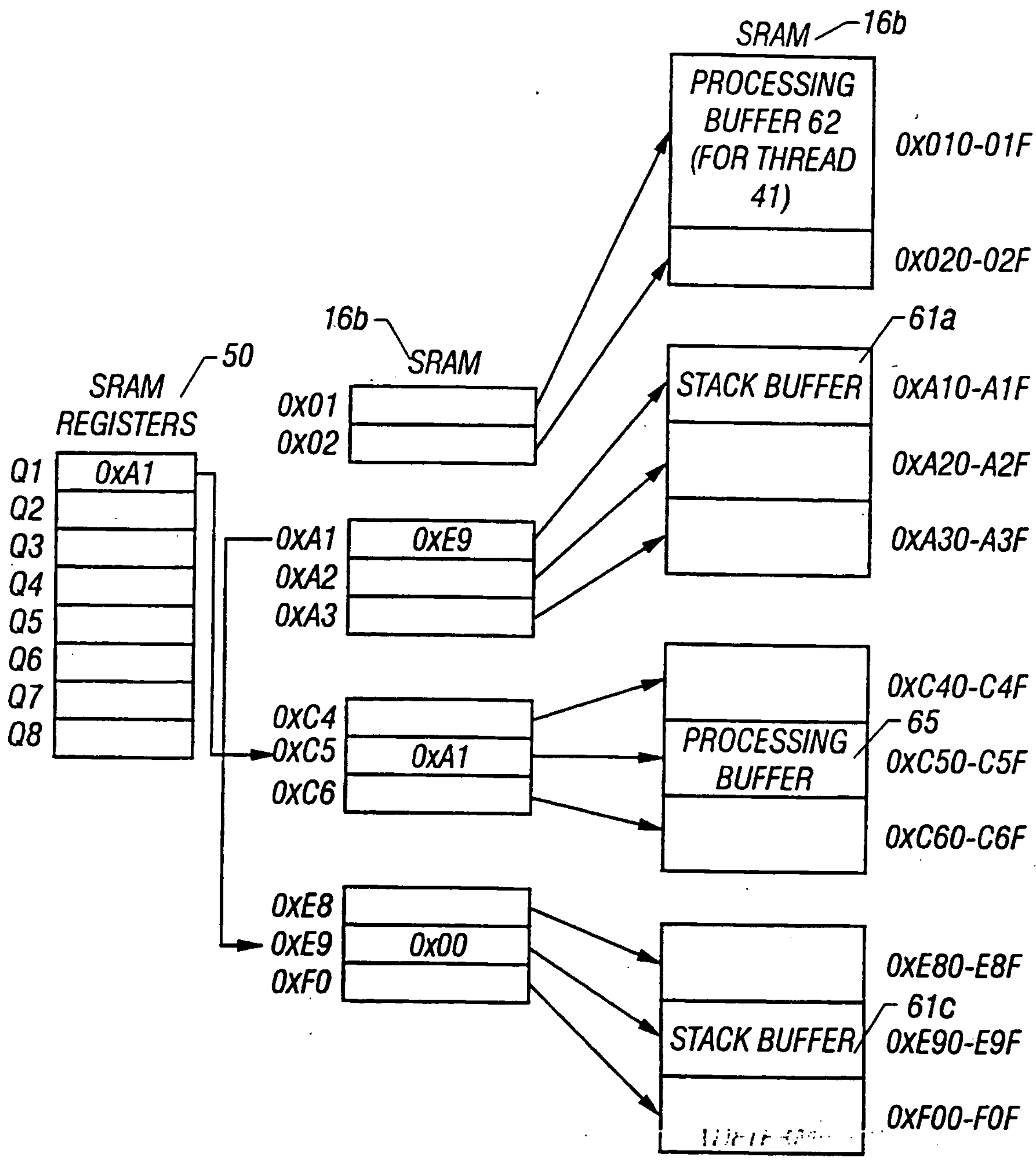

SRAM
16b
PROCESSING BUFFER 62 (FOR THREAD 41)
0x010-01F
0x020-02F
16b
SRAM
61a
STACK BUFFER
0xA10-A1F
0xA20-A2F
0xA30-A3F
50
SRAM REGISTERS
Q1
0xA1
Q2
Q3
Q4
Q5
Q6
Q7
Q8
0x01
0x02
0xA1
0xE9
0xA2
0xA3
0xC4
0xC5
0xA1
0xC6
0xE8
0xE9
0x00
0xF0
0xC40-C4F
65
PROCESSING BUFFER
0xC50-C5F
0xC60-C6F
0xE80-E8F
61c
STACK BUFFER
0xE90-E9F
0xF00-F0F

FIG. 6B

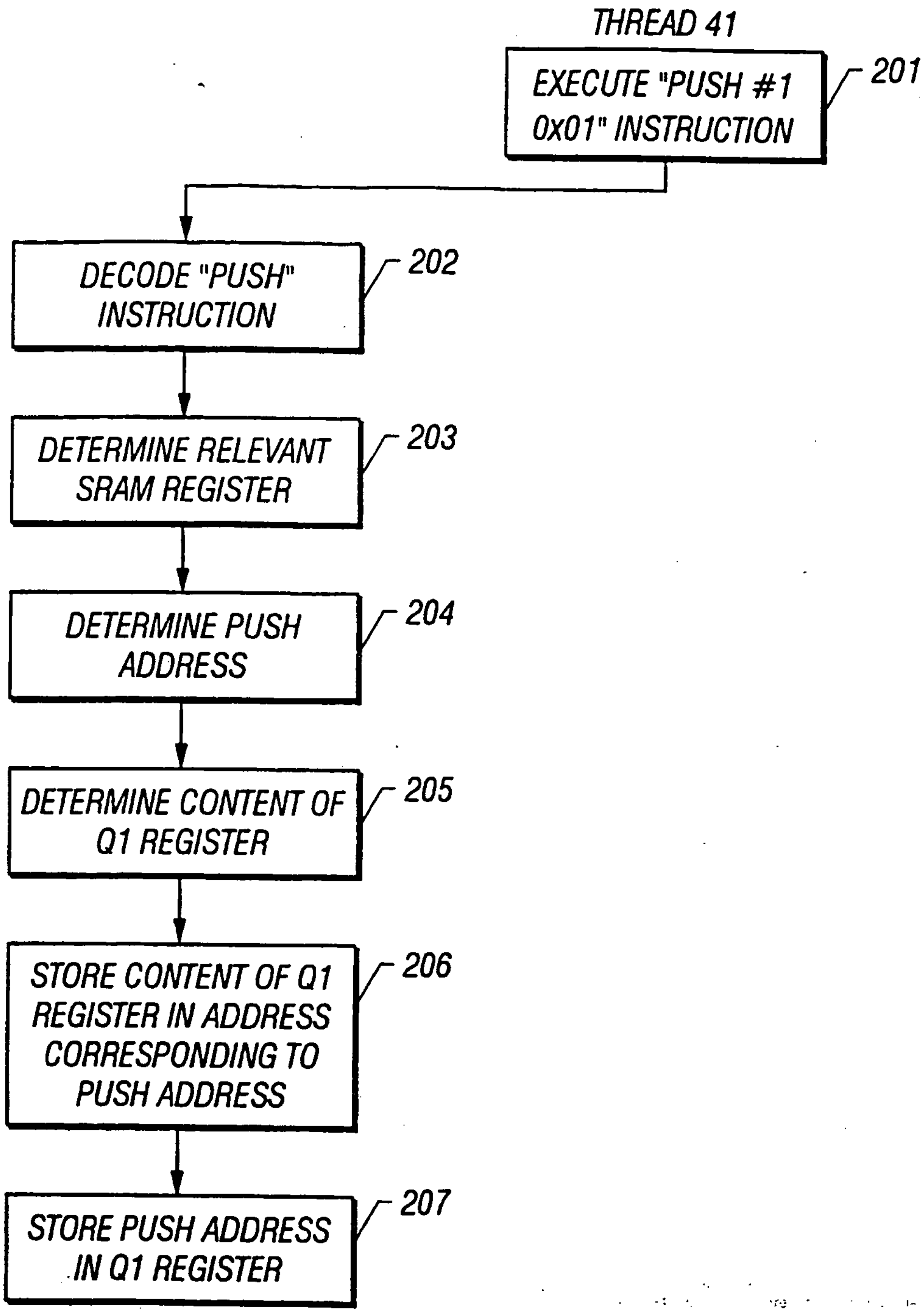

THREAD 41
EXECUTE "PUSH #1 0x01" INSTRUCTION
201
DECODE "PUSH" INSTRUCTION
202
DETERMINE RELEVANT SRAM REGISTER
203
DETERMINE PUSH ADDRESS
204
DETERMINE CONTENT OF Q1 REGISTER
205
STORE CONTENT OF Q1 REGISTER IN ADDRESS CORRESPONDING TO PUSH ADDRESS
206
STORE PUSH ADDRESS IN Q1 REGISTER
207

FIG. 7A

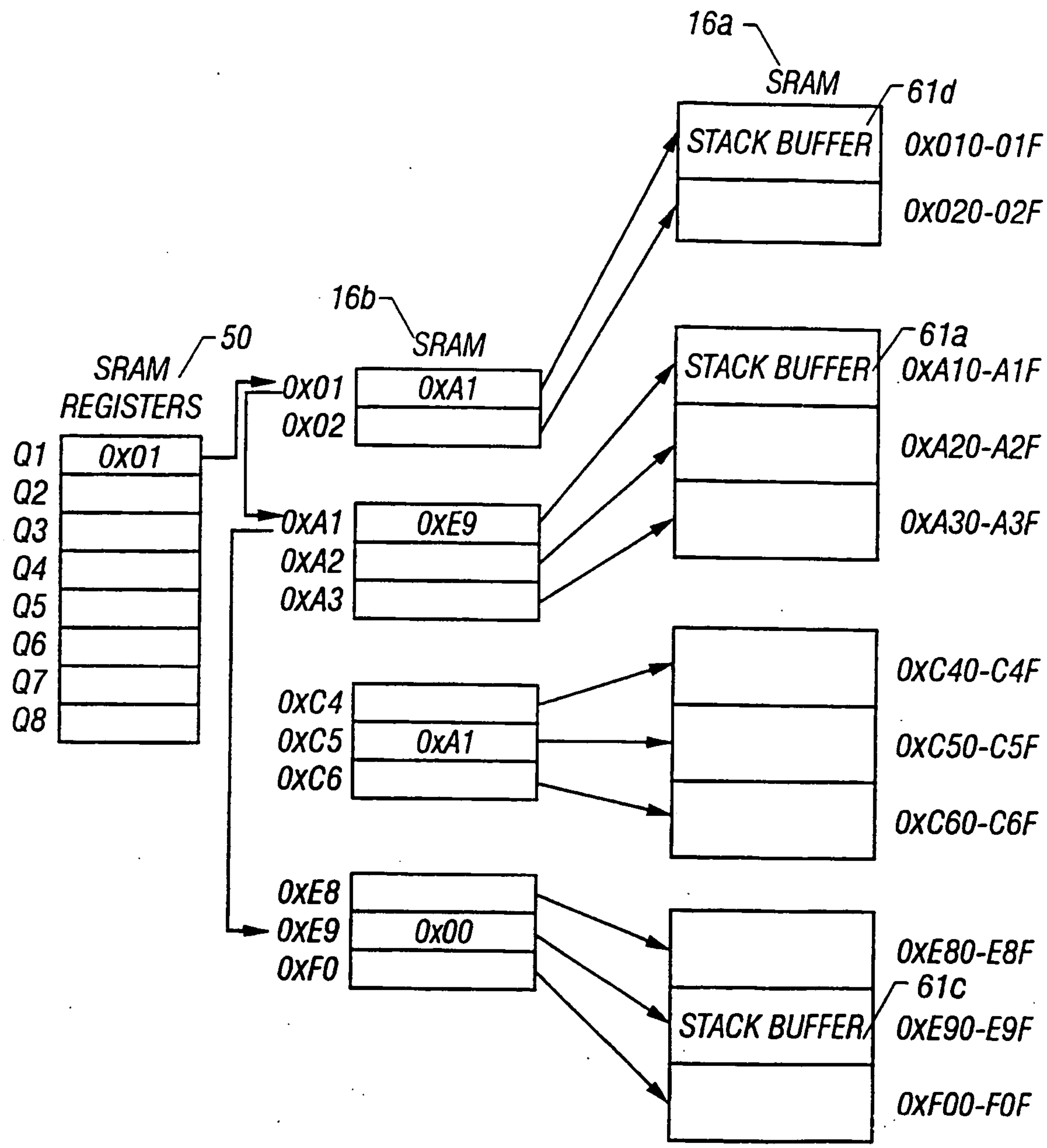
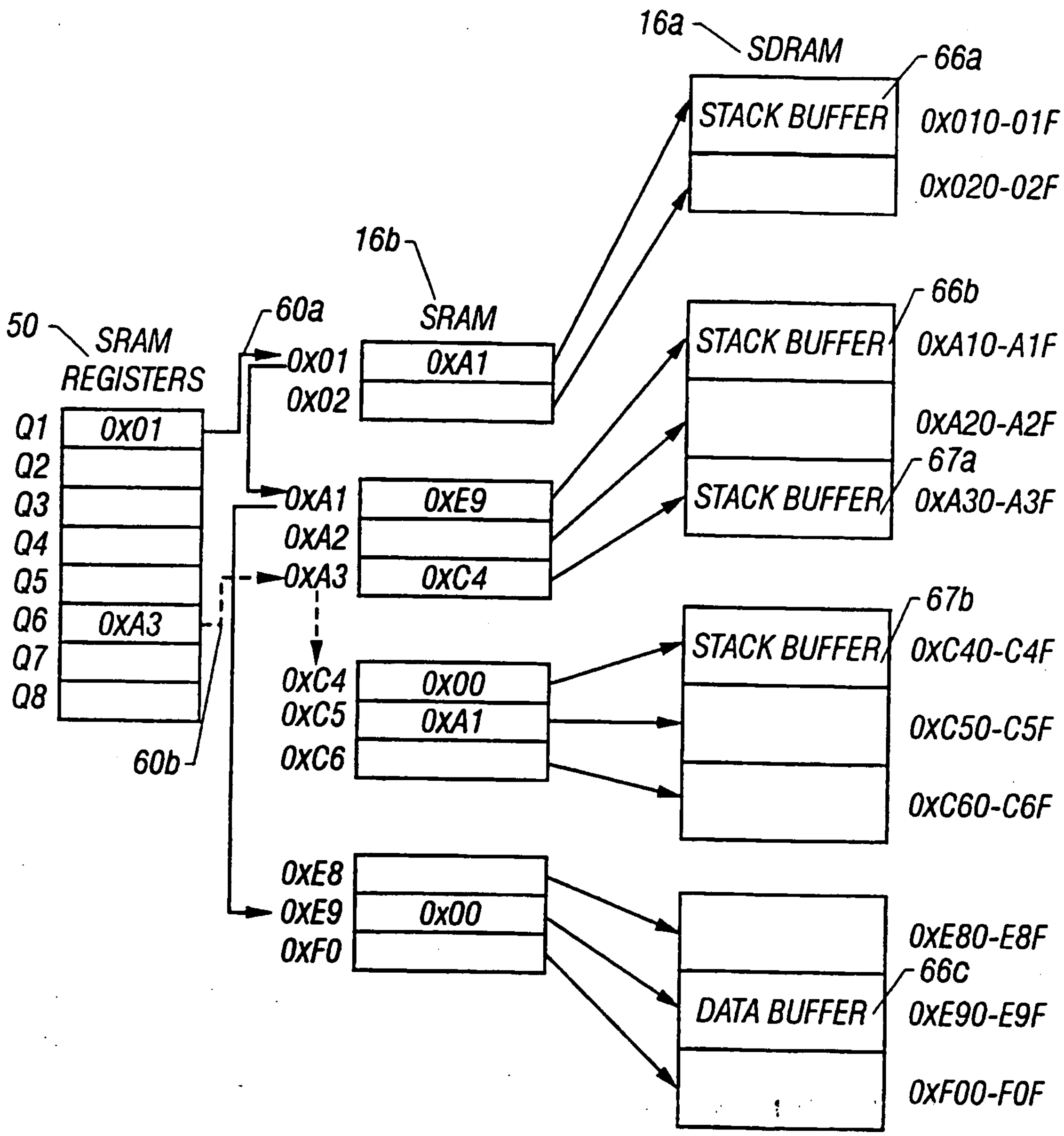

16a
SRAM
61d
STACK BUFFER
0x010-01F
0x020-02F
16b
SRAM
50
SRAM
REGISTERS
61a
STACK BUFFER
0xA10-A1F
0xA20-A2F
0xA30-A3F
Q1
Q2
Q3
Q4
Q5
Q6
Q7
Q8
0x01
0x01
0x02
0xA1
0xA1
0xA2
0xA3
0xE9
0xC4
0xC5
0xC6
0xA1
0xC40-C4F
0xC50-C5F
0xC60-C6F
0xE8
0xE9
0xF0
0x00
0xE80-E8F
61c
STACK BUFFER
0xE90-E9F
0xF00-F0F

FIG. 7B



16a
SDRAM
66a
STACK BUFFER
0x010-01F
0x020-02F
16b
SRAM
66b
STACK BUFFER
0XA10-A1F
0XA20-A2F
67a
STACK BUFFER
0XA30-A3F
67b
STACK BUFFER
0XC40-C4F
0XC50-C5F
0XC60-C6F
0XE80-E8F
66c
DATA BUFFER
0XE90-E9F
0XF00-F0F
50
SRAM REGISTERS
60a
Q1 0X01
Q2
Q3
Q4
Q5
Q6 0XA3
Q7
Q8
60b
0x01 0XA1
0x02
0XA1 0XE9
0XA2
0XA3 0XC4
0XC4 0X00
0XC5 0XA1
0XC6
0XE8
0XE9 0X00
0XF0

FIG. 8

(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
12 July 2001 (12.07.2001)

PCT

(10) International Publication Number
WO 01/50247 A3

(51) International Patent Classification[7]: G06F 9/30, 9/46

(21) International Application Number: PCT/US00/34537

(22) International Filing Date: 19 December 2000 (19.12.2000)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
09/479,377 5 January 2000 (05.01.2000) US

(63) Related by continuation (CON) or continuation-in-part (CIP) to earlier application:
US 09/479,377 (CON)
Filed on 5 January 2000 (05.01.2000)

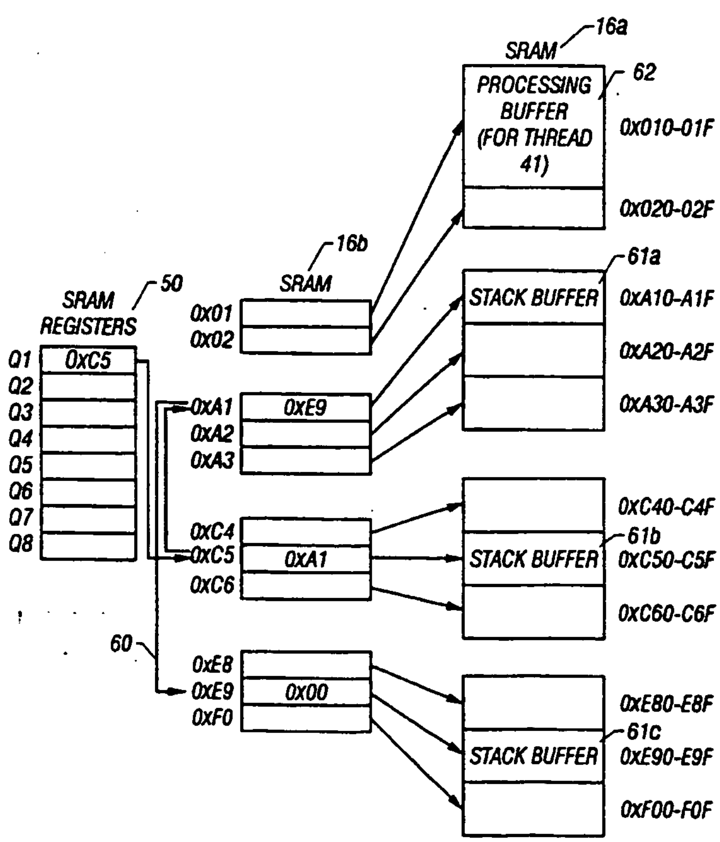
(71) Applicant (*for all designated States except US*): INTEL CORPORATION [US/US]; 2200 Mission College Boulevard, Santa Clara, CA 95052 (US).

(72) Inventors; and
(75) Inventors/Applicants (*for US only*): WOLRICH, Gilbert [US/US]; 4 Cider Mill Road, Framingham, MA 01701 (US). ADILETTA, Matthew, J. [US/US]; 20 Monticello Drive, Worcester, MA 01603 (US). WHEELER, William [US/US]; 9 Darlene Drive, Southborough, MA 01772 (US). CUTTER, Daniel [US/US]; 14 Walnut Street, Townsend, MA 01469 (US). BERNSTEIN, Debra [US/US]; 443 Peakham Road, Sudbury, MA 01776 (US).

(74) Agent: HARRIS, Scott, C.; Fish & Richardson P.C., Suite 500, 4350 La Jolla Village Drive, San Diego, CA 92122 (US).

[Continued on next page]

(54) Title: MEMORY SHARED BETWEEN PROCESSING THREADS



(57) Abstract: A method includes pushing a datum onto a stack by a first processor and popping the datum off the stack by a second processor.

WO 01/50247 A3

(81) Designated States *(national)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CR, CU, CZ, DE, DK, DM, DZ, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, TZ, UA, UG, US, UZ, VN, YU, ZA, ZW.

(84) Designated States *(regional)*: ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE, TR), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GW, ML, MR, NE, SN, TD, TG).

Published:
— *with international search report*

(88) Date of publication of the international search report:
31 January 2002

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

INTERNATIONAL SEARCH REPORT

International Application No
PCT/US 00/34537

A. CLASSIFICATION OF SUBJECT MATTER
IPC 7 G06F9/30 G06F9/46

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC 7 G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)

EPO-Internal, INSPEC

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category *	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	US 5 905 889 A (WILHELM JR GEORGE WILLIAM) 18 May 1999 (1999-05-18)	1,10-13, 29
Y	column 4, line 53 - line 67	14-19, 21-23, 27,30
A	column 7, line 16 - line 35 column 8, line 20 - line 30 column 8, line 60 -column 9, line 32 ---	5-8
Y	EP 0 809 180 A (SEIKO EPSON CORP) 26 November 1997 (1997-11-26) page 3, line 54 - line 55 ---	14-19, 21-23, 27,30
	-/--	

[X] Further documents are listed in the continuation of box C.

[X] Patent family members are listed in annex.

* Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
6 July 2001	06/08/2001

Name and mailing address of the ISA	Authorized officer
European Patent Office, P.B. 5818 Patentlaan 2 NL - 2280 HV Rijswijk Tel. (+31-70) 340-2040, Tx. 31 651 epo nl, Fax: (+31-70) 340-3016	Moraiti, M

2

Form PCT/ISA/210 (second sheet) (July 1992)

International Application No
PCT/US 00/34537

C.(Continuation) DOCUMENTS CONSIDERED TO BE RELEVANT

Category *	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	HYDE R L: "Overview of memory management" BYTE, APRIL 1988, USA, vol. 13, no. 4, pages 219-225, XP002162801 ISSN: 0360-5280 figure 1	1,14,21, 29,30

2

INTERNATIONAL SEARCH REPORT

Information on patent family members

International Application No
PCT/US 00/34537

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 5905889 A	18-05-1999	US 6233630 B	15-05-2001
EP 0809180 A	26-11-1997	JP 10091443 A	10-04-1998